EPFL

Thèse n° 9938

Acceleration of graph pattern mining and applications to financial crime

Présentée le 31 août 2023

Faculté informatique et communications
Laboratoire d'architecture des processeurs
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Jovan BLANUŠA

Acceptée sur proposition du jury

Prof. B. Falsafi, président du jury
Prof. P. Ienne, Dr K. Atasu, directeurs de thèse
Prof. Arvind, rapporteur
Prof. H. Fu, rapporteur
Prof. A.-M. Kermarrec, rapporteuse

École
polytechnique
fédérale
de Lausanne

2023

To my dear Sandra.

Abstract

Various forms of real-world data, such as social, financial, and biological networks, can be represented using graphs. An efficient method of analysing this type of data is to extract subgraph patterns, such as cliques, cycles, and motifs, from graphs. For instance, finding cycles in financial graphs enables the detection of financial crimes such as money laundering and circular stock trading. In addition, extracting cliques from social network graphs enables the detection of communities and could help predict the spread of epidemics. However, extracting such patterns can be time-consuming, especially in larger graphs, because the number of patterns can grow exponentially with the graph size. Therefore, fast and scalable parallel algorithms are required to make the enumeration of these subgraph patterns tractable for real-world graphs.

This thesis presents fast parallel algorithms for the enumeration of maximal cliques and simple cycles. We focus on accelerating the asymptotically-optimal sequential algorithms for enumerating the aforementioned patterns by parallelising them on manycore CPUs. To enable scalable parallelisation of clique and cycle enumeration algorithms, the algorithms presented in this thesis rely on fine-grained parallelisation, in which recursive calls are executed independently of each other using several software threads. However, simply applying the fine-grained parallelisation method to the aforementioned asymptotically-optimal algorithms leads to suboptimal solutions. Parallelising maximal clique enumeration using this method results in increased overhead caused by multithreaded memory management and task scheduling, as well as increased dynamic memory usage. In addition, the asymptotically-optimal algorithms for simple cycle enumeration rely on strict depth-first traversal of their recursion tree, making the fine-grained parallelisation of these algorithms challenging. This thesis addresses these problems and presents parallel algorithms that lead to an almost linear scaling of performance with the number of CPU cores utilised. As a result, the parallel algorithms presented in this thesis are able to achieve an order of magnitude speedup compared to the state-of-the-art parallel algorithms when executed on manycore CPU systems.

To demonstrate the applicability of our accelerated algorithms, this thesis presents the *Graph Feature Preprocessor* library, which can be used to detect financial crime. This library expands

the feature set of financial transactions by enumerating well-known money laundering and financial fraud subgraph patterns, such as simple cycles, in financial transaction graphs. When used in combination with gradient-boosting-based machine learning models, the expanded feature set produced by the library enables significant improvements in prediction accuracy for the problems of money laundering and phishing detection. Furthermore, the efficiency of the subgraph pattern mining algorithms presented in this thesis enables this library to operate in real time.

As financial fraud schemes become more complex, fast algorithms that can detect suspicious behaviour are required. This thesis demonstrates that the parallel graph pattern mining algorithms introduced here can be used to enable fast and accurate detection of such suspicious behaviour.

Key words: Graph pattern mining, maximal clique enumeration, cycle enumeration, graph feature extraction, scalable parallelisation, financial crime detection

Zusammenfassung

Verschiedene Formen von realen Daten, beispielsweise soziale, finanzielle und biologische Netzwerke, können mit Hilfe von Graphen dargestellt werden. Eine effiziente Methode zur Analyse solcher Daten ist die Extraktion von Subgraph-Patterns (Muster), wie Cliquen, Zyklen und Motiven, in Graphen. Das Erkennen von Zyklen in Finanzgraphen ermöglicht beispielsweise die Aufdeckung von Finanzverbrechen wie Geldwäsche und zirkulärem Aktienhandel. Darüber hinaus erlaubt die Extraktion von Cliquen aus Graphen sozialer Netzwerke die Erkennung von Gemeinschaften und könnte helfen, die Ausbreitung von Epidemien vorherzusagen. Die Extraktion dieser Muster kann jedoch zeitaufwändig sein, insbesondere bei grösseren Graphen, da die Anzahl der Muster mit der Grösse des Graphen exponentiell zunehmen kann. Daher werden schnelle und skalierbare parallele Algorithmen benötigt, um die Aufzählung dieser Subgraph-Muster für reale Graphen überschaubar zu machen.

In dieser Arbeit werden schnelle parallele Algorithmen für die Aufzählung von maximalen Cliquen und einfachen Zyklen vorgestellt. Wir konzentrieren uns darauf, die asymptotisch optimalen sequentiellen Algorithmen für die Aufzählung der oben genannten Muster zu beschleunigen, indem wir sie auf Manycore-CPUs parallelisieren. Um eine skalierbare Parallelisierung von Algorithmen zur Aufzählung von Cliquen und Zyklen zu ermöglichen, basieren die in dieser Arbeit vorgestellten Algorithmen auf einer feinkörnigen Parallelisierung. Dabei werden rekursive Aufrufe unabhängig voneinander über mehrere Software-Threads ausgeführt. Die einfache Anwendung der feinkörnigen Parallelisierungsmethode auf die oben beschriebenen asymptotisch optimalen Algorithmen führt jedoch zu suboptimalen Lösungen. Die Parallelisierung der maximalen Cliquenaufzählung mit dieser Methode führt zu erhöhtem Overhead durch die Multithread Speicherverwaltung und die Aufgabenplanung. Zusätzlich wird auch der Verbrauch des dynamischen Speichers erhöht. Darüber hinaus beruhen die asymptotisch optimalen Algorithmen für die einfache Zyklusaufzählung auf einer strikten Depth-First Traversierung des Rekursionsbaums, was die feinkörnige Parallelisierung dieser Algorithmen schwierig macht. Die vorliegende Arbeit befasst sich mit diesen Problemen und stellt parallele Algorithmen vor, die zu einer nahezu linearen Skalierung der Leistung dieser Algorithmen mit der Anzahl der genutzten CPU-Kerne führen. Infolgedessen sind die hier vorgestellten parallelen Algorithmen bei der Ausführung auf Manycore-CPU-Systemen, im

Vergleich zu den modernsten parallelen Algorithmen, in der Lage, eine Beschleunigung von einer Grössenordnung zu erzielen.

Um die Anwendbarkeit unserer beschleunigten Algorithmen zu demonstrieren, wird in dieser Arbeit die Graph Feature Preprocessor-Bibliothek vorgestellt, die zur Erkennung von Finanzkriminalität verwendet werden kann. Diese Bibliothek erweitert das Feature Set von Finanztransaktionen durch Aufzählung bekannter Geldwäsche- und Betrugs-Subgraphenmuster, wie z. B. einfache Zyklen, in Finanztransaktionsgraphen. In Kombination mit maschinellen Lernmodellen, welche auf Gradient-Boosting basieren, ermöglicht die von der Bibliothek erzeugte erweiterte Merkmalsmenge eine erhebliche Verbesserung der Vorhersagegenauigkeit bei der Erkennung von Geldwäsche und Phishing. Darüber hinaus ermöglicht die Effizienz der in hier vorgestellten Algorithmen den Einsatz dieser Bibliothek in Echtzeit.

Da Finanzbetrugsfälle immer komplexer werden, sind schnelle Algorithmen erforderlich, welche verdächtiges Verhalten erkennen können. Diese Arbeit zeigt, dass die hier vorgestellten parallelen Algorithmen zum Graph-Pattern-Mining eine schnelle und genaue Erkennung von verdächtigem Verhalten ermöglichen können.

Stichwörter: Graph-Pattern-Mining, maximale Cliquenaufzählung, Zyklusaufzählung, Graph-Feature-Extraktion, skalierbare Parallelisierung, Erkennung von Finanzkriminalität

Acknowledgements

I would like to express my sincere gratitude to all those who have supported me throughout my PhD journey and who made this thesis possible.

First of all, I would like to express my deepest gratitude to my PhD advisors, Dr. Kubilay Atasu and Prof. Paolo Ienne, for their guidance, support, and encouragement throughout my studies. Their expertise and knowledge have been invaluable in helping me complete my thesis. I am also thankful for their constructive feedback and suggestions, which have been crucial in improving my research skills. I am especially grateful to Kubilay for inviting me to come to IBM Research in Zürich, where we have worked closely for the past few years, and for all of his hard work and patience with me while I was learning how to conduct research and write scientific papers. I am particularly grateful to Paolo for his continued involvement in my supervision and for providing valuable help even though I was not physically present at EPFL and this PhD project is outside of his area of expertise. I was very fortunate to be advised by these two brilliant researchers.

I would like to thank the members of my thesis jury, Prof. Arvind, Prof. Babak Falsafi, Prof. Haohuan Fu, and Prof. Anne-Marie Kermarrec, for taking the time to read and evaluate my thesis. Their valuable feedback and questions helped me greatly improve my thesis. Additionally, I would like to thank the Swiss National Science Foundation for providing financial support for my PhD project under project number 172610.

My PhD research was carried out at IBM Research in Zürich, where I met many brilliant people. It was a great pleasure to collaborate with Dr. Erik Altman, Dr. Andreea Anghel, Maximo Cravero, Béni Egressy, Luc von Niederhäusern, Dr. Thomas Parnell, Dr. Radu Stoica, and Dr. Haris Pozidis, who was also my manager at IBM. I would also like to thank the rest of my team at IBM, Dr. Dionysios Diamantopoulos, Gosia Lazuka, Dr. Jan Van Lunteren, Dr. Nikolaos Papandreou, Dr. Roman Pletka, and Dr. Slaviša Sarafijanović, for lunches, coffees, and discussions. Special thanks to my office mates, Dr. Corey Lammie and Thanos Vasilopoulos, for keeping me company during the thesis writing. I am also grateful to Dr. Robert Haas for welcoming me to the Hybrid Cloud department and to Judith Blanc for

administrative assistance. In addition, I would like to thank Anne-Marie Cromack and Linda Rudin from the IBM Publications team for taking their time to proofread my manuscripts and translate the abstract of this thesis into German. I am truly grateful for the opportunity to be part of this amazing organisation, and I look forward to new research challenges.

I am also grateful to the past and current members of the Processor Architecture Laboratory at EPFL, Dr. Mikhail Asiatici, Louis Coulon, Ayatallah Elakhras, Andrea Guerrieri, Prof. Lana Josipović, Sahand Kashani, Dr. Stefan Nikolić, Lucas Ramirez, Mohamed Shahawy, and Canberk Sönmez, who made me feel like I was a part of the lab, even though I was not physically there. Special thanks to Chantal Schneeberger, who was always willing to assist me with EPFL-related administrative matters. In addition, I would like to thank Prof. Yusuf Leblebici, who was briefly my thesis supervisor, for welcoming me to his former lab at EPFL at the beginning of my PhD studies.

Lastly, I would like to thank my mother Sanja, father Mile, brother Ljuban, and sister Jelena for their support and love. In addition, I would like to thank my dear wife Sandra for her patience, love, and encouragement. She has been beside me since before I came to Switzerland, and her endless support and care gave me the strength to overcome obstacles during my studies. I am grateful to have her in my life.

Zürich, July 12, 2023 J. B.

Contents

List of Figures

List of Tables

1 Introduction

The data-driven society in which we live today requires efficient mechanisms to manage, integrate, and analyse large volumes of data. Graphs are a widely adopted tool for representing various types of data and are used in many different domains [Mat17; Wan+20a; Noe+16; Web21]. A graph consists of a set of vertices connected by edges, where an edge typically represents a relationship or an interaction between two vertices. For example, financial transaction graphs represent a set of bank accounts and transactions, where each transaction connects two accounts [Mat17]; social graphs contain individuals and their relationships [LRU14]; and the WWW consists of web pages connected with hyperlinks. As graph data can be complex and difficult to understand, we require algorithms that can efficiently extract useful information from graphs. Detection of patterns and correlations, classification of graph objects, and discovery of nontrivial new relationships between graph objects in a scalable manner are key capabilities of modern data analytics platforms.

Graph pattern mining represents a set of methods used for analysing graph data by extracting subgraphs with a given property from a graph [CH06; AW10]. Extracting such subgraphs enables uncovering hidden relationships in a graph, which helps to better understand the underlying data. Examples of subgraphs that could be extracted from a graph include clusters [For10], motifs [Ahm+15; PBL17], the most frequent subgraph patterns that appear in the graph [JCZ13; Els+14; Abd+16], or subgraphs that match a certain predefined pattern. Predefined subgraph patterns include cliques (i.e., complete subgraphs) [BK73], cycles (i.e., closed paths) [MD76], bicliques (i.e., complete bipartite subgraphs) [Epp94], or arbitrary user-defined patterns [HLL13; Cor+04; SL20]. Figure 1.1 depicts the three maximal cliques that can be found in the graph shown. This thesis investigates the possibility of efficiently executing algorithms for finding maximal cliques and simple cycles on modern manycore CPUs and their potential applications in the financial domain.

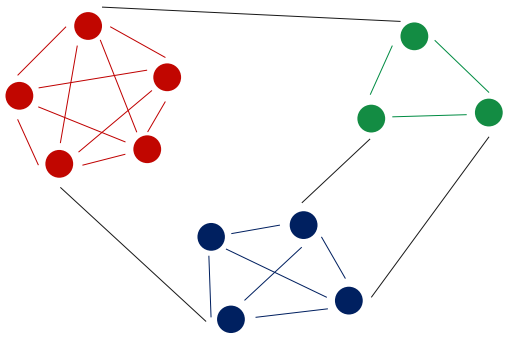


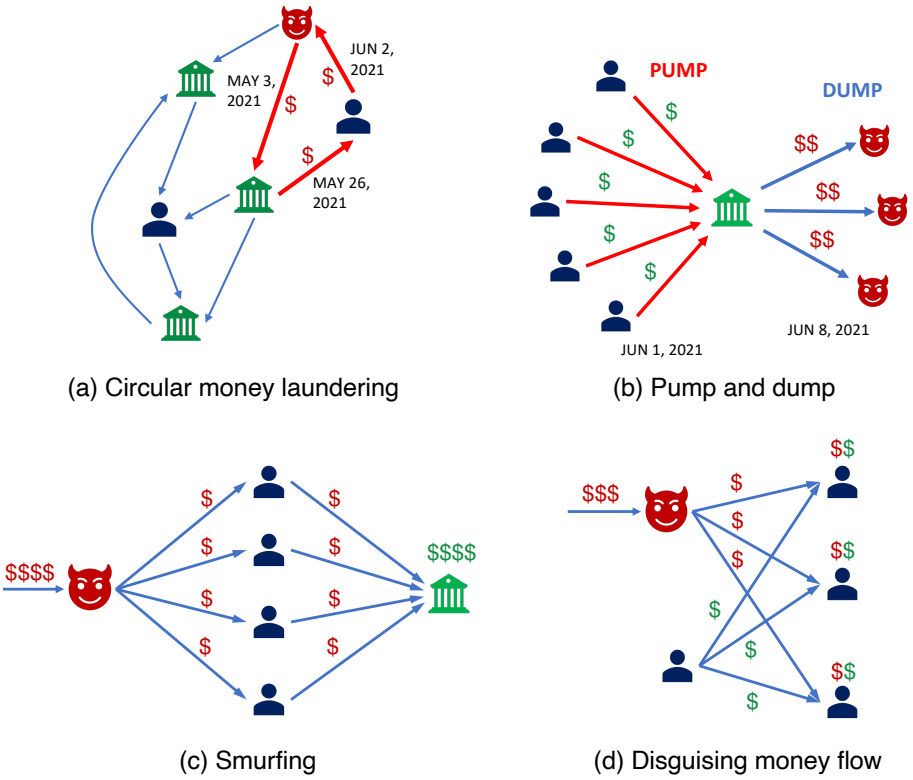
Figure 1.1: Mining all maximal cliques with more than two vertices. In this graph, three such cliques exist and are annotated with different colours. In a social network, each such maximal clique could indicate a community of individuals that all know and interact with each-other.

1.1 Graph Pattern Mining Applications

Graph pattern mining has applications in many domains, such as biology, social network analysis, and finance. Maximal cliques can be used to detect communities in social networks [LWN18], as shown in Figure 1.1. Furthermore, maximal cliques can be used to predict protein functions in protein interaction networks [YZT14; Yu+06] and to predict how epidemics spread [Dan+11]. Finding simple cycles is useful in electronic design automation to detect combinatorial loops, which are typically forbidden in electronic circuits [GS05; PD21]. In a software bug tracking system, a dependency between two software bugs requires one bug to be addressed before the other [SAS21]. Circular bug dependencies are undesirable and can be detected by finding simple cycles. Other applications include detecting feedback loops in biological networks [KC07; KK09] and detecting unstable relationships in social networks [GRW17; Zho+18]. Furthermore, subgraph patterns can be used to improve the accuracy of graph neural netowrks by providing additional information about the graph [Bou+23; Bar+21].

This thesis focusses on the applications of subgraph pattern enumeration to the financial domain, more specifically, to the detection of financial crime. The term “financial crime” refers to unlawful acts committed for financial gain, such as money laundering, tax evasion, and credit card fraud [NKL21]. In financial transaction graphs, where the vertices represent accounts and the edges transactions, a cycle is a strong indicator of financial crime, such as money laundering, tax avoidance [HK20; SK21], and credit card fraud [Qiu+18]. Finding cycles also enables the detection of *circular trading*, which is a financial crime used to manipulate stock prices. In circular trading, a group of traders tries to artificially increase the price of a stock by circulating its shares among themselves [PA08; Isl+09; Jia+13]. As a result, it appears that the market has increased interest in this stock, which raises the price of its shares. A depiction of a cycle in a financial transaction network is given in Figure 1.2a.

The *pump and dump* scheme is another method used to manipulate stock prices and is illustrated in Figure 1.2b. In this scheme, a group of malicious traders buy stocks in a company and use social media or other publicity to attract other traders to invest in this company, which increases (“pumps”) the stock price of this company. After the stock price increases



(a) Circular money laundering (b) Pump and dump

(c) Smurfing (d) Disguising money flow

Figure 1.2: Crime patterns in financial transaction graphs. Red nodes denote malicious accounts performing financial crimes. Green and red dollar signs denote licit and illicit funds, respectively.

sufficiently, malicious traders sell ("dump") the stocks. Other traders realise that the value of the stock is artificially inflated after its value drops and typically suffer financial losses [NKL21]. In financial transaction graphs, this scheme is characterised by a *gather-scatter* pattern in which a vertex has high fan-in and fan-out. Similar patterns can be observed for the accounts on the Ethereum blockchain that perform phishing scams [Che+19a].

Another type of money laundering is *smurfing* [KM11; Cor23; RT04; Lee+20; Li+20; Sta+21], which is illustrated in Figure 1.2c. In this case, an entity, represented as a red node in Figure 1.2c, attempts to insert illicit funds obtained from illegal activities into the legal banking system. To avoid raising suspicion, this entity uses multiple intermediaries ("smurfs") to deposit a portion of these illicit funds into a bank account. The amount of money that each intermediary deposits is below the reporting threshold; therefore, banks will be less likely to find such a transaction suspicious and investigate it. In financial transaction graphs, smurfing manifests itself as a subgraph pattern shown in Figure 1.2c, which has a source vertex (red) connected with several intermediate vertices (blue), and these intermediate vertices are connected to a target vertex (green). We refer to this subgraph pattern as a *scatter-gather* [SK21].

A more complex subgraph pattern in a financial transaction graph that could indicate illegal activities is a *directed biclique* [SK21; Web+18]. This pattern, illustrated in Figure 1.2d, is a

bipartite subgraph in which all the vertices of one partition are connected with all the vertices of the other partition [LSL06; Pri00]. The existence of such a pattern in a financial transaction graph could be associated with disguising the trail of money, where illicit funds (red dollar signs in Figure 1.2d) are combined with legally obtained funds (green dollar signs in Figure 1.2d). As a result, the source of illicit funds is obfuscated.

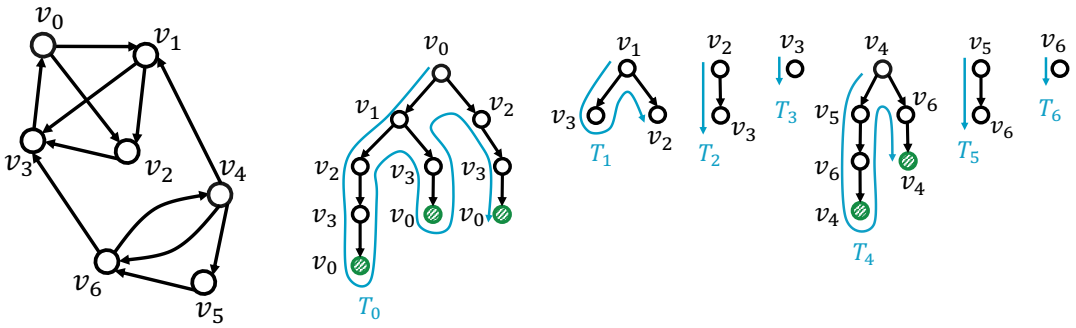
In cryptocurrency transaction networks, criminals use sophisticated mixing and shuffling schemes to cover the trace of their activities [Liu+21a]. Such schemes can usually be formulated in terms of subgraph structures, such as scatter-gather patterns and bicliques [Che+19b; Ron+21; Wal21]. The discovery of such suspicious subgraph patterns enables the identification of criminal activities and their perpetrators. Furthermore, other subgraph patterns, such as temporal motifs [PBL17], can be used to detect these cryptocurrency mixing services [Wu+21].

In general, subgraph patterns in the financial graph can represent various schemes used in financial crime. Therefore, timely detection of such fraudulent patterns helps minimise the damage caused by criminals.

1.2 Challenges of Accelerating Graph Pattern Mining

One of the main drawbacks of algorithms for mining subgraph patterns is their high algorithmic complexity. These algorithms usually have significantly higher algorithmic complexity [JCZ13; AW10] compared to other types of graph processing algorithms [MWM15; Bat+15; Cor09a]. Graph processing algorithms, such as Breadth-First Search [Cor09b], PageRank [Pag+98], Single-Source Shortest Path [Dij59; Bel58; Joh77], and Strongly-Connected Components [Tar72; FHP00], have a time complexity that is upper bound by a polynomial of graph parameters with a low degree. On the other hand, the time complexity of graph pattern mining algorithms is upper bound by an exponential of a graph parameter [TTT06; BK73; HM20; Els+14; Car+18], such as the number of vertices n, maximum degree Δ_{max}, or degeneracy d, i.e., the smallest value such that each nonempty subgraph of a graph has a vertex with at most d edges [LW70]. This exponential complexity of graph pattern mining algorithms usually leads to long execution times.

The reason for the high algorithmic complexity of the graph pattern mining algorithms is the large number of subgraph patterns that might exist in a graph. In theory, a graph with n vertices may have up to $3^{n/3}$ maximal cliques [MM65], up to $\frac{1}{3^{1/3}-1}3^{n/3}$ maximal bicliques [GKL12], and up to $\sum_{k=2}^{n}\binom{n}{k}(k-1)!$ simple cycles [All85] (e.g., in a complete directed graph). Even sparse graphs, which have a low degeneracy value d, might have a high number of subgraph patterns. Theoretically, it has been proven that the number of maximal cliques and maximal bicliques in sparse graphs can be C^d times higher than the number of vertices in a graph [ELS13; Epp94], where $C \geq 3^{1/3}$ is a constant. Furthermore, planar graphs, which have a degeneracy of at most five [LW70], have been shown to have exponentially many simple cycles in terms of n [AT08; AFK97; Buc+07]. Thus, fast algorithms and their parallel implementations are required to make graph pattern mining algorithms tractable.



(a) Example graph (b) Recursion tree for v_0 (c) Recursion trees for other starting vertices

Figure 1.3: The example graph (a) and the recursion trees (b),(c) generated during the search for cycles starting from different vertices. Each recursion tree node is marked with a vertex that the associated recursive call visits. A recursion tree node marked with green reports a simple cycle. To prevent reporting duplicate cycles, the algorithm visits only the vertices with indices greater than the index of the starting vertex. The recursion trees perform different amounts of work, which results in an imbalanced workload across threads $T_0, \ldots, T_6$.

The most efficient sequential graph pattern mining algorithms that search for subgraph patterns usually rely on recursive search and exploration [TTT06; ELS13; Joh75; LSL06; SL20; Els+14; DBS18]. These algorithms maintain a subgraph that matches a part of a pattern the algorithm is enumerating (e.g., cycle or clique). An algorithm updates this subgraph at each recursive call by inserting a new vertex into that subgraph such that it still matches a part of the pattern. Once this subgraph matches the entire pattern, the algorithm reports it. For example, recursive algorithms for cycle enumeration [Joh75; Tie70] that search for cycles in a graph shown in Figure 1.3a starting from vertex v_0 would maintain a path that starts with v_0 as indicated in the recursion tree shown in Figure 1.3b. This path is recursively updated until vertex v_0 is inserted again and the path becomes a simple cycle (see green nodes in Figure 1.3b). In addition, these recursive graph pattern mining algorithms usually start a search for subgraph patterns from each vertex of a graph or a subset of its vertices, as illustrated in Figures 1.3b and 1.3c. However, the unpredictable shape and size of these recursion trees and their dynamic construction make these algorithms challenging to efficiently parallelise.

One method for parallelising the recursive graph pattern mining algorithms is to execute the recursion trees for each starting vertex independently using several threads. This parallelisation method is illustrated in Figures 1.3b and 1.3c. This method can be implemented using existing vertex-centric processing frameworks, such as Pregel [Mal+10], Giraph [Ave11], and others [MWM15]. We refer to this method as *coarse-grained parallelisation* of graph pattern mining algorithms. The main drawback of coarse-grained parallelisation is that the workload across the recursion trees can be severely imbalanced. For instance, the thread T_0 exploring the recursion tree shown in Figure 1.3b performs significantly more work compared to the other threads shown in Figure 1.3c. Furthermore, for real-world graphs, which often exhibit a power-law or a log-normal distribution of vertex degrees [BP16; BC19], the execution time of

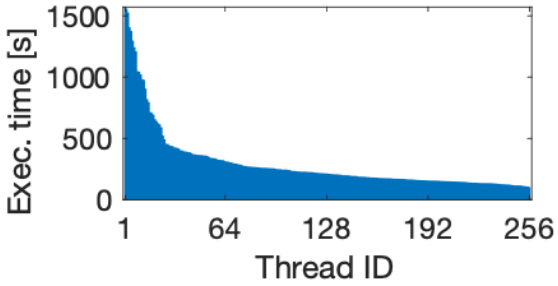


Figure 1.4: Per-thread execution time of the coarse-grained parallel Johnson algorithm that searches for simple cycles in in the *wiki-talk-temporal* [LK14] graph within a $12h$ time window. A significant workload imbalance can be observed as the execution time of the longest-running thread is several times higher than the execution time of the majority of other threads.

recursive graph pattern mining algorithms can be dominated by searches that start from a small set of vertices. This behaviour leads to a workload imbalance, as shown in Figure 1.4, and limits the scalability of parallel implementations.

This thesis focusses on the acceleration of subgraph pattern enumeration problems using manycore CPUs for two types of subgraph patterns: maximal cliques and simple cycles. Even though the state-of-the-art algorithms for enumerating these two types of patterns have a similar structure, they present different acceleration opportunities and challenges. The maximal clique enumeration algorithm by Eppstein et al. [ELS13] and the algorithms it extends [BK73; TTT06] rely on recursive search and can be parallelised using the aforementioned coarse-grained parallelisation, which has limited scalability. Additionally, the vertex-set intersections that are performed in each recursive call of these algorithms dominate the execution time of the algorithms. Similar challenges are also present in other sequential graph pattern mining algorithms, such as subgraph isomorphism [Cor+04; HLL13; SL20; HZY18], frequent pattern discovery [MW19], biclique enumeration [LSL06; Che+22], and k-clique listing [DBS18]. For instance, vertex-set intersections are frequently used in graph pattern mining algorithms to determine how a vertex is connected with other vertices from the subgraph, as shown in Figure 1.5. Thus, the methods presented in this thesis for accelerating maximal clique enumeration can also be used to accelerate these related graph pattern mining algorithms.

The limited scalability issue with coarse-grained parallelisation of the maximal clique enumeration can be addressed by executing a single recursion tree using several threads. This solution is possible because a recursion tree of the asymptotically-optimal algorithm for maximal clique enumeration [ELS13; TTT06] can be executed in any order, making its recursive calls independent. On the other hand, each recursion tree generated by the asymptotically-optimal algorithm for simple cycle enumeration by Johnson [Joh75] is required to be executed in a strict depth-first order, making it difficult to explore the recursion tree using several threads without significant loss of efficiency. Similar behaviour can be observed in algorithms that extend the Johnson algorithm [Joh75] to enable the enumeration of simple cycles under temporal-ordering [KC18] and hop [Pen+19] constraints. Furthermore, in contrast to maximal clique enumeration algorithms, simple cycle enumeration algorithms do not require

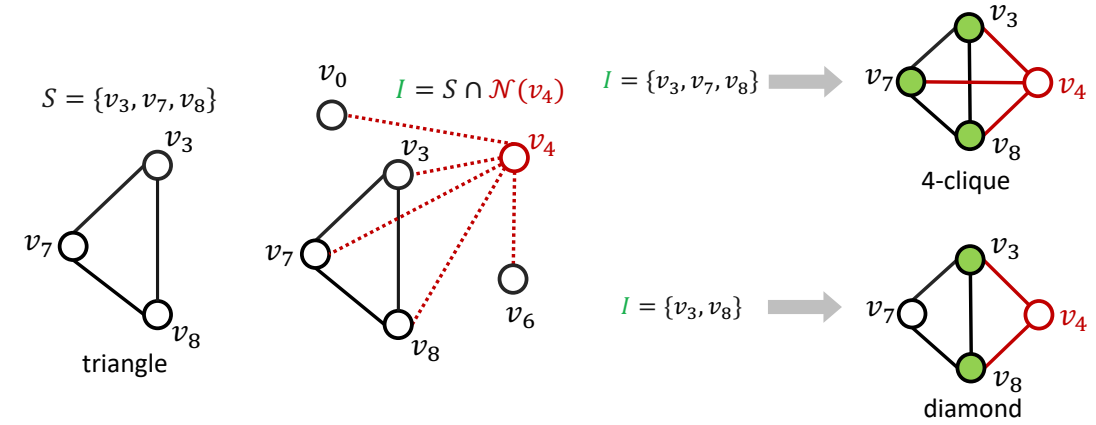


Figure 1.5: Example of how vertex-set intersections can be used in graph pattern mining. When extending the current subgraph S from (a) with the vertex v_4, set intersection is used to check how v_4 is connected with the rest of the vertices in the existing subgraph S, as shown in (b). Based on the intersection result, we can determine the type of the extended subgraph, as illustrated in (c).

vertex-set intersections because, when searching for simple cycles, the operation of extending the subgraph with a new vertex, shown in Figure 1.5, is not required to check how that vertex is connected with the rest of the subgraph. As a result, the recursive calls of simple cycle enumeration algorithms are computationally lightweight, as they mostly perform pointer chasing; hence, the opportunity to exploit data parallelism in these algorithms is limited.

In this thesis, we show how the aforementioned challenges have been addressed to enable fast and scalable algorithms for maximal clique and simple cycle enumeration algorithms. We focus on the acceleration of these algorithms using shared-memory manycore CPUs. Such CPUs enable the concurrent execution of many software threads that can easily communicate with each other thanks to the shared memory. As a result, dynamic load balancing techniques [BL99] can be used to address the workload imbalance and the limited scalability of the coarse-grained parallelisation approach. In addition, these CPUs often support vector instructions that can be used to accelerate set intersection operations that occur in maximal clique enumeration algorithms.

Another option for accelerating graph pattern mining is to use GPUs [CA22; Alm+22; Alm22; Che+20a; JMV20; Guo+20]. A GPU commonly contains several *streaming multiprocessors*, where each streaming multiprocessor consists of multiple lightweight cores and a memory block shared by those cores. A collection of threads called the *thread block* is executed on a single streaming multiprocessor, and each thread block is divided into *warps* consisting of several threads. Threads of the same warp run the most efficiently when they are executing the same instruction that is accessing consecutive memory locations. If that is not the case, the performance of the GPU might be degraded [ROA13]. Even though this problem could be

alleviated in maximal clique enumeration by using threads from the same warp to perform a set intersection in parallel [Alm+22; Alm22], this problem would be more significant in simple cycle enumeration algorithms because the data parallelism in these algorithms is limited. Furthermore, workload imbalance is more prominent in GPUs compared to CPUs because GPUs have significantly more cores. To address this problem, the workload of threads executed on the same streaming multiprocessor can be dynamically balanced, but a workload imbalance may still occur between different streaming multiprocessors [Alm22]. Another method for alleviating the workload imbalance that occurs in GPUs when executing graph pattern mining algorithms is to provide more parallelism by traversing the recursion trees of these algorithms in a breadth-first-search order [JMV20; Che+20a]. However, this approach significantly increases the memory required for storing the intermediate results of the graph pattern mining algorithms, and thus might be limited to smaller graphs [CA22]. Therefore, we do not consider using GPUs for this thesis.

1.3 Thesis Statement and Contributions

This thesis explores the possibility of accelerating graph pattern mining algorithms using modern manycore CPUs. It seeks to address the parallelisation challenges described in Section 1.2 and show that the state-of-the-art sequential algorithms for maximal clique and simple cycle enumeration can be parallelised in a scalable manner. By addressing these challenges, the thesis shows that parallel graph mining algorithms can be used to enable applications in financial crime detection introduced in Section 1.1.

The statement of this thesis is formulated as follows.

> *Graph pattern mining algorithms can be effectively accelerated using the existing manycore CPUs, which enables fast detection of financial crime.*

To support this thesis, three major contributions are presented.

First, the thesis presents a fast parallel implementation of the state-of-the-art sequential maximal clique enumeration (MCE) algorithm [Bla+20a; Bla+20b]. As mentioned in Section 1.2, this sequential MCE algorithm [BK73; ELS13] performs time-consuming set intersection operations in each recursive call and is challenging to parallelise in a scalable manner. We address the former challenge by implementing hash-join-based and merge-join-based set intersections using vector instructions available in modern manycore CPUs. The comprehensive theoretical analysis performed concluded that the MCE algorithm that uses hash-join-based set intersections is either faster or uses less memory compared to the MCE algorithm that uses merge-join-based set intersections. Thus, the fast MCE implementation presented in this thesis accelerates set intersections by using a vectorised hash-join-based set intersection implementation. To parallelise this algorithm in a scalable manner, each thread executes a subset of recursion calls, enabling several threads to execute a single recursion tree, thus addressing the problem of a load imbalance that exists when the algorithm is parallelised

using the coarse-grained method (see Section 1.2). We implement such a *fine-grained parallel* approach using a shared-memory parallel processing framework [VAR19] and address the resulting performance overheads to further accelerate this algorithm and reduce its memory footprint. The resulting parallel MCE implementation is capable of processing graphs with tens of millions of vertices and up to two billion edges in just a few minutes on a single CPU.

Second, the thesis introduces scalable parallel simple cycle enumeration algorithms [BIA22; BAI23] that are based on the asymptotically-optimal sequential algorithms by Johnson [Joh77] and by Read and Tarjan [RT75]. Although the Johnson algorithm is faster than the Read-Tarjan algorithm in practice [Gro16; MD76], it is also more challenging to parallelise due to the requirement that its recursion trees have to be executed in a strict depth-first-search order (see Section 1.2). To enable scalable parallelisation of the Johnson algorithm, we have relaxed its strictly depth-first-search-based exploration, which enables a recursion tree of this algorithm to be executed by several threads in parallel. In addition, we demonstrate that the Read-Tarjan algorithm does not have the same limitations as the Johnson algorithm and is, thus, easier to parallelise in a scalable manner. Furthermore, we show that our method for scalable parallelisation of the Johnson algorithm can be adapted to parallelise state-of-the-art algorithms for enumerating cycles under temporal [KC18] and hop constraints [Pen+19]. The proposed parallel algorithms for simple cycle enumeration are scalable in both theory and practice and are an order of magnitude faster than the algorithms parallelised using the coarse-grained parallelisation method discussed in Section 1.2.

Finally, the thesis presents a graph-based feature extraction library called *Graph Feature Preprocessor*. This software library extracts the well-known money laundering and fraud patterns in financial transaction graphs, which are used to enrich the feature set of the financial transactions. In addition, it supports a dynamic in-memory multigraph data structure that enables fast dynamic updates and efficient detection of suspicious subgraph patterns using fine-grained parallelism. Furthermore, a graph machine learning pipeline is developed for monitoring financial transaction graphs that uses the Graph Feature Preprocessor to expand the feature set of the financial transactions and gradient-boosting machine learning models to predict suspicious transactions. As a result, this pipeline enables up to 64% improvement in the minority-class F1 score of money laundering detection tasks while achieving throughput rates of up to 100′000 transactions per second using only 6 CPU cores.

1.4 Thesis Organisation

This thesis is organised as follows. Chapter 2 presents the background information required for this thesis, including the notation, the parallel programming model used, and the state-of-the-art sequential graph pattern mining algorithms. Fast parallel algorithms for the enumeration of maximal cliques and simple cycles are introduced in Chapters 3 and 4, respectively. Chapter 5 presents the graph-based feature extraction library called *Graph Feature Preprocessor* that generates features based on graph patterns, such as cycles and scatter-gather patterns.

This chapter also demonstrates that our library can enable fast and accurate detection of financial transactions associated with financial crime, such as money laundering and phishing. Conclusions and potential future research directions are discussed in Chapter 6.

2 Preliminaries and Background

This chapter introduces the main concepts and algorithms used throughout the thesis. First, we define the graph notation and graph pattern mining problems addressed in this thesis. Then, we describe the parallel programming model used throughout the thesis. Next, we introduce the basic concepts that this thesis uses to analyse the parallel algorithms presented. Finally, we introduce several sequential graph pattern mining algorithms on which this work is based. The notation used is given in Table 2.1.

2.1 Graphs

A **directed graph** is denoted as $\mathcal{G}(\mathcal{V},\mathcal{E})$ and is defined with a set of vertices $\mathcal{V}$ and a set of directed edges $\mathcal{E} = \{u \rightarrow v \mid u, v \in \mathcal{V}\}$. A directed edge $u \rightarrow v$ is defined by its *source* vertex u and its *target* (*destination*) vertex v. An outgoing edge of a given vertex v is defined as $v \rightarrow w$, and an incoming edge of v is defined as $u \rightarrow v$, where $v \rightarrow w, u \rightarrow v \in \mathcal{E}$. The set of *outgoing neighbours* of a given vertex v in the graph $\mathcal{G}$ is defined as $\mathcal{N}_{\mathcal{G}}(v) = \mathcal{N}^{+}_{\mathcal{G}}(v) = \{\forall\, w \mid\ v \rightarrow w \in \mathcal{E}\}$, and the set of *incoming neighbours* of a vertex v in the graph $\mathcal{G}$ is defined as $\mathcal{N}^{-}_{\mathcal{G}}(v) = \{\forall\, w \mid w \rightarrow v \in \mathcal{E}\}$. A graph $\mathcal{G}$ is **undirected** if $\mathcal{N}^{+}_{\mathcal{G}}(v) = \mathcal{N}^{-}_{\mathcal{G}}(v)$ for every $v \in \mathcal{V}$, in which case we simply use $\mathcal{N}_{\mathcal{G}}(v)$ to denote the set of neighbours of a vertex v. In this thesis, we omit the subscript $\mathcal{G}$ from $\mathcal{N}^{+}_{\mathcal{G}}(v)$, $\mathcal{N}^{-}_{\mathcal{G}}(v)$, and $\mathcal{N}_{\mathcal{G}}(v)$ if it is clear to which graph we are referring; for example, only one graph is being considered. A **temporal graph** $\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{T})$ is a directed graph that has its edges annotated with timestamps [PBL17], where for each edge $e \in \mathcal{E}$, there exists a timestamp $t(e) \in \mathcal{T}$. We refer to an edge annotated with a timestamp as a *temporal edge*. In this thesis, a temporal edge with a source vertex u, a target vertex v, and a timestamp t is denoted as $(u \rightarrow v, t)$. Temporal graphs are often *multigraphs* [Bal97], i.e., graphs that may have several edges with the same source and destination vertices. Such edges with the same source and destination vertices are referred to as *parallel edges*.

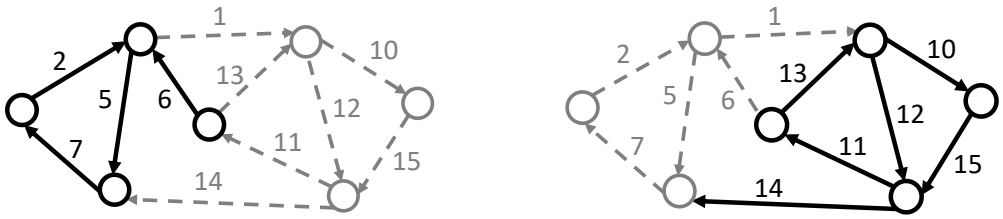
In a directed graph, the *out-degree* and the *in-degree* of a vertex v are defined as the number of outgoing and incoming edges of v, respectively. For instance, the out-degree of the vertex v_4 from the graph shown in Figure 1.3 is three, and its in-degree is one. In the case of an

Table 2.1: Summary of the notation used in the thesis.

Symbol	Description
General notation	
$\mathcal{G}(\mathcal{V},\mathcal{E})$	Graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$.
n, e	Number of vertices and edges in a graph.
$\mathcal{N}(v), \mathcal{N}_{\mathcal{G}}(v), \mathcal{N}^{+}_{\mathcal{G}}(v)$	The set of (outgoing) neighbours of v.
$\mathcal{N}^{-}(v), \mathcal{N}^{-}_{\mathcal{G}}(v)$	The set of incoming neighbours of v.
$u \rightarrow v$	A directed edge from u to v.
$(u \rightarrow v, t_{uv})$	A temporal directed edge from u to v with a timestamp t_{uv}.
$\Delta_{max}, \Delta_{avg}$	Maximum and average degree of a graph.
d, h	The degeneracy and an h-index of a graph.
c, s	Number of simple cycles and maximal simple paths in a graph.
$\lvert\mathcal{X}\rvert$	Number of elements in the data structure $\mathcal{X}$.
$[t_{w1} : t_{w2}]$	Time window between timestams t_{w1} and t_{w2}.
δ	Size of a time window.
Parallel algorithms	
p	Number of threads used by a parallel algorithm.
T_i for $i = 0, 1, 2\ldots$	The i-th thread executing a parallel algorithm.
$T_p(n)$	Execution time of a parallel algorithm using p threads.
$T_\infty(n)$	Depth of a parallel algorithm.
$W_p(n)$	Amount of work a parallel algorithm performs using p threads.
$\mathcal{X}_{T_i}$	Data structure $\mathcal{X}$ is maintained by the thread T_i.
Maximal clique enumeration	
R	The current clique of the BK algorithm.
P, X	The candidate vertex, and the exlcude set of the BK algorithm.
ρ, χ	Sizes of sets P and X, respectively.
$H_{P,X}$	Subgraph created using P and X as shown in Eppstein et al [ELS13].
Cycle enumeration	
Π	Current simple path explored by cycle enumeration algorithm.
Blk	Set of blocked vertices of cycle enumeration algorithm.
Blist	Unblock list of the Johnson algorithm.
E	Path extension of the Read-Tarjan algorithm.

undirected graph, the out-degree of a vertex v is equal to its in-degree and is commonly referred to as the *degree* of v. In this thesis, the maximum degree of a vertex in a graph is denoted as Δ_{max}, and the average degree is denoted as Δ_{avg}. The *degeneracy* of a graph represents the smallest value d, such that each nonempty subgraph of a graph has a vertex with a degree at most d [LW70]. An *h-index* of a graph h is a maximum value such that a graph contains h vertices of degree at least h [ES12]. For example, for the undirected graph shown in Figure 1.1, Δ_{max}, Δ_{avg}, d, and h are, respectively, 5, 3.83, 4, and 4. Note that the aforementioned parameters Δ_{max}, Δ_{avg}, d, and h can be used to express the sparsity of a graph.

The main subgraph patterns explored in this thesis are cliques, cycles, and scatter-gather patterns. A *clique* is defined for an undirected graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ as a set of vertices $W \subset \mathcal{V}$ such



(a) Time window [2: 7] (b) Time window [10: 15]

Figure 2.1: Two snapshots of a temporal graph associated with two different time windows of size $\delta = 5$. The solid arrows indicate the edges that belong to the respective time windows.

that there exists an edge in $\mathcal{E}$ between every pair of vertices in W. A **maximal clique** is a clique that is not fully contained in another clique [BK73]. An example of 3 maximal cliques is given in Figure 1.1. A *path*, defined for an undirected or directed graph, between the vertices v_0 and v_k, denoted as $v_0 \rightarrow v_1 \ldots \rightarrow v_k$, is a sequence of vertices such that there exists an edge between every two consecutive vertices of the sequence. A *simple path* is a path without repeated vertices. A simple path is *maximal* if the last vertex of the path has no neighbours or all its neighbours are already in the path [EG59]. A **cycle** is a path of non-zero length from a vertex v to the same vertex v. A **simple cycle** is a cycle without repeated vertices except for the first and last vertices. For simplicity, the term "simple" is often omitted when referring to simple cycles and simple paths in this thesis. The number of maximal simple paths and the number of simple cycles in a graph are denoted as s and c, respectively (see Table 2.1). Note that s can be exponentially larger than c [Tar73]. A path or a cycle is said to satisfy **hop-constraint** L if the number of edges on that path or cycle is less than or equal to L. In temporal graphs, a **temporal cycle** is a simple cycle in which the edges appear in increasing order of their timestamps. A **scatter-gather** pattern, illustrated in Figure 1.2c, is defined for a directed graph using a starting vertex, an end vertex, and a set of intermediate vertices such that there exists an edge from the starting edge to each intermediate vertex and an edge from each intermediate vertex to the end vertex.

A subgraph of a temporal graph occurs within a **time window** $[t_{w1} : t_{w2}]$ if every edge of that subgraph has a timestamp t_s such that $t_{w1} \leq t_s \leq t_{w2}$. Figure 2.1 shows the simple cycles of a temporal graph that occur within two different time windows of size $\delta = 5$. This graph contains one simple cycle in the time window [2 : 7] (Figure 2.1a), which is also a temporal cycle, and two simple cycles in the time window [10 : 15] (Figure 2.1b), neither of which is a temporal cycle.

This thesis focusses on the following enumeration problems. **Maximal clique enumeration** is the problem of finding all maximal cliques of an undirected graph $\mathcal{G}$. **Simple cycle enumeration** is the problem that requires finding all simple cycles of a directed or undirected graph $\mathcal{G}$. The goal of **temporal cycle enumeration** is to find all temporal cycles of a temporal graph $\mathcal{G}$. Finally, if a hop constraint is imposed on simple cycles, the simple cycle enumeration problem becomes **hop-constrained simple cycle enumeration**. This thesis focusses on cycle

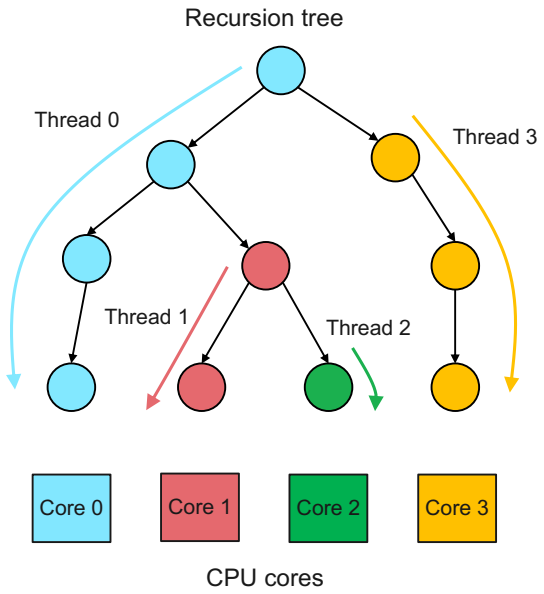


Figure 2.2: Parallel depth-first traversal of a recursion tree. Each thread is pinned to a CPU core and executes a part of the recursion tree in a depth-first manner.

enumeration problems under the time-window constraint, in which it is required to find all cycles that occur within a time window of a given size δ. Thus, we consider cycle enumeration problems on temporal graphs.

2.2 Parallel Programming Model

To parallelise the graph pattern mining algorithms in a scalable manner, this thesis uses the *fork-join* parallel model [Con63; MDR12]. In this model, *fork* divides the control flow of a program into two or more flows that can be executed in parallel, and *join* combines these control flows into a single sequential flow. This thesis assumes that *fork* and *join* commands can be nested, which makes this model convenient for parallelising the recursive algorithms explored in this thesis.

This thesis uses the modified Cilk notation [Blu+96a] to express nested fork-join parallelism in algorithms. A *task* is a sequential part of a program that can be executed by a thread independently of other tasks. The *spawn* command creates a new task with the given input arguments and specifies that this task is ready to be executed by a thread. A *parent* task can spawn several *child* tasks. The *depth* of a task represents the number of its direct ancestors. A parent task can wait for its child tasks to finish their execution by executing the *sync* command. The spawn and sync commands represent fork and join, respectively, in the fork-join model. Another method to specify a fork is to use a *parallel foreach* loop, which enables each loop iteration to be executed independently by a different thread. In the algorithms described in this thesis, a task usually consists of a single recursive call of a graph pattern mining algorithm,

and a parallel foreach is used to create the initial task for each vertex or edge of a graph in parallel. The keywords task, spawn, sync, and parallel used in the algorithms presented in this thesis to express nested fork-join parallelism are highlighted in orange.

Nested fork-join parallelism can be implemented using shared-memory parallel processing frameworks, such as Threading Building Blocks (TBB) [Kuk07], Cilk [Blu+96a], OpenMP [Qui04], and Java fork/join framework [Ora22]. In addition to dynamic task creation, these frameworks also have a dynamic task management system that assigns the spawned tasks to the task queues of the available threads. Furthermore, a work-stealing scheduler [BL99; Kuk07; Blu+96a] enables a thread that is not executing a task to *steal* a task from the task queue of another thread. Stealing tasks enables dynamic load balancing and ensures full utilisation of the threads when there are sufficiently many tasks.

For this thesis, the algorithms were implemented mainly using TBB; however, the algorithms presented in this thesis can also be implemented using the other frameworks mentioned. A thread of this framework executes the task that it spawned the last [Kuk07]. If the task queue of a thread is empty, this thread steals the oldest task from the task queue of another thread. As a result, TBB enables parallel depth-first traversal of a recursion tree of a recursive algorithm when a task consists of a recursive call. This parallel depth-first traversal of a recursion tree is illustrated in Figure 2.2. Additionally, TBB offers a scalable memory allocator that reduces the overhead of concurrent memory allocations.

2.3 Analysis of Parallel Algorithms

To analyse parallel algorithms, this thesis uses the *work-depth* model [BM10]. In this model, the algorithms are described using *work*, which is the number of operations a parallel algorithm performs, and *depth*, which is the length of the longest sequence of dependent operations. In addition, we refer to the time to execute a parallel algorithm on a problem of size n using p threads as $T_p(n)$. The size of a graph is determined by the number of vertices n as well as the number of edges m, but for simplicity, we will refer only to n. Assuming that each operation takes unit time to complete, we define work and depth as follows.

Definition 1. *(Work) The work $W_p(n)$ of a parallel algorithm executed on a problem of size n using p threads is $W_p(n) = \sum_{k=0}^{p-1} T_p(n)$.*

Definition 2. *(Depth) The depth $T_\infty(n)$ of a parallel algorithm executed on a problem of size n is $T_\infty(n) = \lim_{p\to\infty} T_p(n)$.*

In this thesis, we use the notions of *work efficiency* and *scalability* to analyse parallel algorithms [BM10]. The *work efficiency* and the *scalability* are formally defined as follows.

Definition 3. *(Work efficiency) A parallel algorithm is work-efficient if and only if $W_p(n) \in O(T_1(n))$.*

Definition 4. *(Scalability) A parallel algorithm is scalable if and only if* $\lim_{n\to\infty}\left(\lim_{p\to\infty}\frac{T_p(n)}{T_1(n)}\right)=0.$

Informally, a work-efficient parallel algorithm performs the same amount of work as its serial version, within a constant factor. Scalability implies that, for sufficiently large inputs, increasing the number of threads increases the speedup of the parallel algorithm with respect to its serial version.

This thesis also uses the notion of *strong scalability*, which is defined as follows [JaJ92].

Definition 5. *(Strong scalability) A parallel algorithm is strongly scalable if and only if* $\frac{T_1(n)}{T_p(n)} = \Theta(p)$ *for large enough* n.

Whereas Definition 4 implies that the speedup $T_1(n)/T_p(n)$ achieved by a parallel algorithm with respect to its serial execution is infinite when the number of threads p is infinite, Definition 5 implies that the speedup is always in the order of p. Another related concept is weak scalability, which requires the speedup to be in the order of p when the input size per thread is constant. Note that both strong scalability and weak scalability imply scalability.

Finally, to derive the worst-case time complexity of work-efficient parallel algorithms, we use Brent's theorem [Bre74].

Theorem 1. *(Brent's theorem) The time complexity of executing a work-efficient parallel algorithm on* p *processors is* $T_p(n) \in O\left(\frac{T_1(n)}{p} + T_\infty(n)\right)$.

2.4 Sequential Graph Pattern Mining Algorithms

In this section, we describe the state-of-the-art sequential algorithms for maximal clique and simple cycle enumeration. We use these algorithms as a starting point for the development of our fast and scalable algorithms for the enumeration of these two types of patterns.

2.4.1 Maximal Clique Enumeration Algorithms

The Bron-Kerbosch (BK) algorithm is one of the most successful algorithms for listing all maximal cliques of a graph [BK73]. It is a recursive algorithm that operates on three sets of vertices during each recursive call: the set R stores the vertices that form the currently largest clique; the candidate set P stores the vertices that may form a clique with the ones from R; and the exclude set X stores the vertices that have already been considered and, therefore, cannot participate in new cliques. The P and X sets contain vertices adjacent to the vertices of R at each step, as illustrated in Figure 2.3, which is ensured by the set intersection. At each recursive call, a vertex v from the set of possible vertices P is added to R, such that R still forms a clique. We then recursively determine whether the extended set R is part of a larger clique or not. After all the cliques that contain the vertex v have been enumerated, the vertex is moved

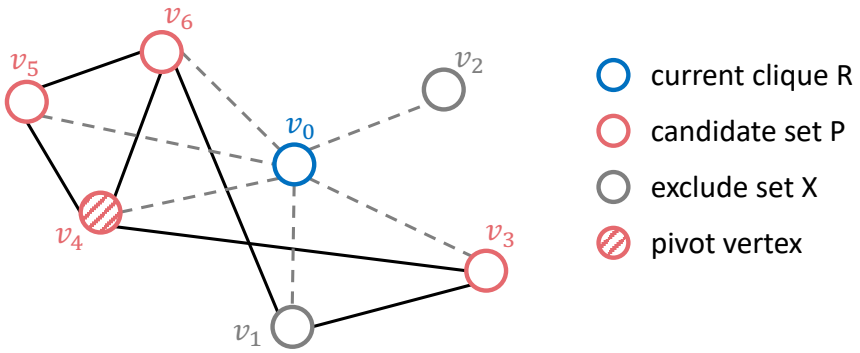


Figure 2.3: A state of the BK algorithm's recursive call showing vertices from R, P, and X, and the edges between those vertices. Starting from the given state, the algorithm would eventually report a clique $\{v_0, v_4, v_5, v_6\}$. However, a clique $\{v_0, v_1, v_3\}$ would not be reported because $v_1 \in X$, indicating that this clique has already been reported.

Algorithm 1: The BK algorithm w. pivoting by Tomita et al [TTT06]

Input: R - set of vertices representing the current clique
P - candidate vertex set
X - exclude vertex set
$\mathcal{G}$ - the input graph

```
1  Function BKPivot (R, P, X, G)
2      if P = ∅ then
3          if X = ∅ then Report R as a maximal clique;
4          return ;
5      pivot = getPivot(P, X, G);
6      foreach v : P \ N_G(pivot) do
7          BKPivot(R + {v}, P ∩ N_G(v), X ∩ N_G(v), G);
8          P = P − {v};
9          X = X + {v};
10 Function getPivot (P, X, G)
       Output: pivot - the pivot vertex
11     foreach v : P ∪ X do
12         t_v = |P ∩ N_G(v)|;
13     return arg max_v(t_v);
```

to the set X. If sets P and X are both empty, R is reported as a maximal clique. At the beginning, sets R and X are empty, and the set P contains all the vertices of the input graph.

Bron and Kerbosch also introduced the pivoting strategy to reduce the number of unnecessary recursive calls. When expanding R, instead of considering all vertices of the set P, a pivot vertex *pivot* from P is chosen, and vertices neighbouring the pivot vertex $\mathcal{N}_G(pivot)$ are not considered for expansion of R. This approach results in a reduction in the number of invoked recursive calls and faster execution of the algorithm. Note that Bron and Kerbosch did not derive the worst-case time complexity of this algorithm; however, it was empirically shown

Algorithm 2: BKDegeneracy ($\mathcal{G}(\mathcal{V},\mathcal{E})$) [ELS13]

Input: $\mathcal{G}$ - the input graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$

```
1 Order vertices V in G using degeneracy ordering;
2 foreach v_i : V do
3 |   P = N_G(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{n-1}};
4 |   X = N_G(v_i) ∩ {v_0, v_1, ..., v_{i-1}};
5 |   R = {v_i};
6 |   BKPivot(R, P, X, G) ;                    ▷ Invoke Algorithm 1
```

that the execution time of this algorithm is proportional to $3.14^{n/3}$ [BK73; TTT06].

Tomita et al. [TTT06] improved the BK algorithm by choosing the pivot vertex from $P \cup X$ in a way that maximises the number of vertices from $\mathcal{N}_G(pivot)$ that are excluded from the expansion of R (i.e., $|P \cap \mathcal{N}_G(pivot)|$ is maximised). For example, in Figure 2.3, the vertex v_4 is selected as a pivot, and $\mathcal{N}_G(v_4) = \{v_3, v_5, v_6\}$ are excluded from the expansion. Thus, the vertex $P \setminus \mathcal{N}_G(v_4) = v_4$ is the only vertex considered for the expansion of R. This approach leads to minimising the number of generated recursive calls and results in a worst-case time complexity of $O(3^{n/3})$. This result is worst-case optimal, given that the worst-case number of cliques in a graph is $O(3^{n/3})$ [CK08; MM65; TTT06]. The BK algorithm with the pivoting strategy introduced by Tomita et al. [TTT06] is shown in Algorithm 1.

Eppstein et al. [ELS10] presented the further improved BK algorithm by imposing the degeneracy ordering when starting the search for cycles. Degeneracy represents the smallest value d, such that each nonempty subgraph of a graph has a vertex with at most d edges. If a graph has degeneracy d, its largest clique can have at most $d+1$ vertices, and the vertices of the graph can be ordered in such a way that each vertex has d or fewer neighbouring vertices that come later in the ordering. The main idea used in the improved algorithm is to compute a degeneracy ordering of the vertices before invoking the original BK algorithm with pivoting, as shown in Algorithm 2. Each loop iteration of Algorithm 2 starts the search for cliques from each vertex v_i of $\mathcal{V}$ by assigning this vertex to the set R. For each vertex v_i, sets P and X are computed as the neighbours of v_i that come later and before in the degeneracy ordering, respectively. For the sets created as described, the BK algorithm with pivoting by Tomita et al. [TTT06] is invoked. Due to degeneracy ordering, every set P generated in lgorithm 2 will have at most d vertices, which limits the depth of each recursion tree. As a result, the worst-case time complexity of the BK algorithm is reduced to $O(dn3^{d/3})$. In the rest of this thesis, we will refer to this algorithm as *the BK algorithm with degeneracy ordering* or simply *the BK algorithm*.

2.4.2 Simple Cycle Enumeration Algorithms

The following algorithms for simple cycle enumeration perform recursive searches to incrementally update simple paths that can lead to cycles. Each algorithm iterates the vertices or edges of the graph and independently constructs a recursion tree to enumerate all the cycles

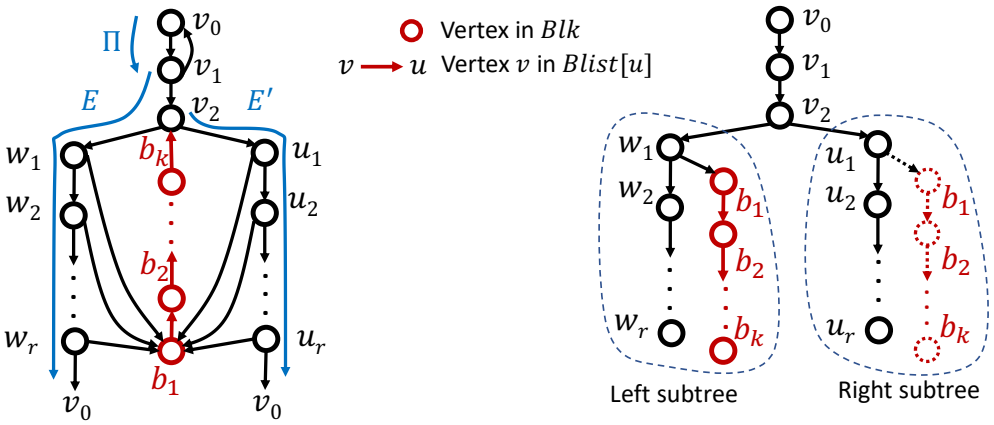


Figure 2.4: (a) An example graph and (b) the recursion tree constructed when searching for cycles that start from v_0. The nodes of the recursion tree represent the recursive calls of the depth-first search. Whereas the Johnson algorithm would visit the vertices in red only once during the exploration of the left subtree, the Read-Tarjan algorithm visits those vertices also in the right subtree, as indicated using dotted lines.

starting from that vertex or edge. The difference between these algorithms is to what extent they reduce the redundant work performed during the recursive search, which we discuss next.

The Tiernan algorithm [Tie70] enumerates simple cycles using a brute-force search. It recursively extends a simple path Π by appending a neighbour u of the last vertex v of Π, provided that u is not already in Π, as shown in Algorithm 3. A clear downside of this algorithm is that it can repeatedly visit vertices that can never lead to a cycle. When searching for cycles in the graph shown in Figure 2.4a starting from the vertex v_0, this algorithm would explore the path that contains $b_1, \ldots, b_k$ $2r$ times. From each vertex w_i and u_i, with $i \in \{1, \ldots, r\}$, the Tiernan algorithm would explore this path only to discover that it cannot lead to a simple cycle. As noted by Tarjan [Tar73], the Tiernan algorithm explores every simple path and, consequently, all maximal simple paths of a graph. Exploring a maximal simple path takes $O(e)$ time because it requires visiting each edge of the graph in the worst case. Given a graph with s maximal simple paths (see Table 2.1), the worst-case time complexity of the Tiernan algorithm is $O(se)$.

The Johnson algorithm [Joh75] improves upon the Tiernan algorithm by avoiding the vertices that cannot lead to simple cycles when appended to the current simple path Π. For this purpose, the Johnson algorithm maintains a set of blocked vertices *Blk* that are avoided during the search. In addition, a list of vertices *Blist*[w] is stored for each blocked vertex w. Whenever a vertex w is unblocked (i.e., removed from *Blk*) by the Johnson algorithm, the vertices in *Blist*[w] are also unblocked. This unblocking process is performed recursively until no more vertices can be unblocked, which we refer to as the *recursive unblocking* procedure. The pseudocode of the Johnson algorithm is given in Algorithm 4, and the recursive unblocking procedure is shown in lines 17–21 of that algorithm.

Algorithm 3: TiernanBacktrack (v, Π, $\mathcal{G}$) [Tie70]

Input: v - the current vertex
$\mathcal{G}$ - the input graph
InOut: Π - the current path

```
1 Π.push(v);
2 v0 = Π.front();                                  ▷ The starting vertex
3 foreach u : N_G(v) s.t. u.id > v0.id do          ▷ Recursively explore the neighbours of v
4 |   if u = v0 then
5 |   |   report cycle Π;
6 |   else if u ∉ Π then                           ▷ Make sure the path is simple
7 |   |   TiernanBacktrack(u, Π, G);               ▷ Create a child recursive call
8 Π.pop()
```

A vertex v is blocked (i.e., added to *Blk*) when visited by the algorithm. If a cycle is found after recursively exploring every neighbour of v that is not blocked, the vertex v is unblocked. However, v is not immediately unblocked if no cycles are found after exploring its neighbours. Instead, the *Blist* data structure is updated to enable unblocking of v in a later step by adding v to the list $Blist[w]$ of every neighbour w of v. This delayed unblocking of the vertices enables the Johnson algorithm to discover each cycle in $O(e)$ time in the worst case. Because this algorithm requires $O(n+e)$ time to determine that there are no cycles, its worst-case time complexity is $O((n+e)(c+1))$ [SL76]. Note that because s can be exponentially larger than c [Tar73], the Johnson algorithm is asymptotically faster than the Tiernan algorithm.

In the example shown in Figure 2.4a, every simple path Π that starts from v_0 and contains vertices $b_1, \ldots, b_k$ is a maximal simple path, and, thus, it cannot lead to a simple cycle. The Johnson algorithm would block $b_1, \ldots, b_k$ immediately after visiting this sequence once and then keep these vertices blocked until it finishes exploring the neighbours of v_2. As a result, the Johnson algorithm visits vertices $b_1, \ldots, b_k$ only once, rather than $2r$ times the Tiernan algorithm would visit them. Note that because these vertices get blocked during the exploration of the left subtree of the recursion tree, they are not going to be visited again during the exploration of the right subtree. Effectively, a portion of the right subtree is pruned (see the dotted path in Figure 2.4b) based on the updates made on *Blk* and *Blist* during the exploration of the left subtree. This strictly sequential depth-first exploration of the recursion tree is critically important for achieving a high pruning efficiency, but it also makes scalable parallelisation of the Johnson algorithm extremely challenging, which we are going to cover in Section 4.3.

The Johnson algorithm can be adapted to efficiently search for temporal cycles [KC18] and hop-constrained simple cycles [Pen+19]. Adapting the Johnson algorithm to search for cycles under different constraints is further explored in Section 4.5 of this thesis.

The Read-Tarjan algorithm [RT75] also has a worst-case time complexity of $O((n+e)(c+1))$. The pseudocode of this algorithm is shown in Algorithm 5. This algorithm maintains a current path Π between a starting vertex and a frontier vertex. A recursive call of this algorithm iterates

Algorithm 4: JohnsonBacktrack (v, $\mathcal{G}$, Π, Blk) [Joh77]

Input: v - the current vertex; $\mathcal{G}$ - the input graph
InOut: Π - the current path; Blk - Blocked vertex set; Blist - Blocked list
Output: *true* if a cycle was found

```
1  v0 = Π.front();                                   ▷ The starting vertex
2  Π.push(v); Blk = Blk ∪ {v};
3  found = false;
4  foreach u : N_G(v) s.t. u.id > v0.id do          ▷ Recursively explore the neighbours of v
5  |  if u = v0 then
6  |  |  report cycle Π;
7  |  |  found = true;
8  |  else if u ∉ Blk then
9  |  |  f = JohnsonBacktrack(u, G, Π, Blk) ;       ▷ Create a child recursive call
10 |  |  found = found ∨ f;
11 Π.pop();
12 if found then                                     ▷ Unblock vertices if a cycle was found
13 |  RecursiveUnblock(v, Blk, Blist);               ▷ Defined in lines 17-21
14 else
15 |  foreach u : N_G(v) do Blist[u] = Blist[u] ∪ {v};
16 return found;
17 Function RecursiveUnblock (u, Blk, Blist)
18 |  Blk = Blk/{u};
19 |  foreach w : Blist[u] do
20 |  |  Blist[u] = Blist[u]/{w};
21 |  |  RecursiveUnblock(w, Blk, Blist);
```

the neighbours of the current frontier vertex and performs a depth-first search (DFS). Assume that v_0 is the starting vertex and v_1 is the frontier vertex of Π (see Figure 2.4a). From each neighbour $y \in \{v_0, v_2\}$ of v_1, a DFS tries to find a path extension E back to v_0 that would form a simple cycle when appended to Π. In the example shown in Figure 2.4a, the algorithm finds two path extensions, one indicated as E and one that consists of the edge $v_1 \rightarrow v_0$. The algorithm then explores each path extension by iteratively appending the vertices from it to the path Π. For each vertex x added to Π, the algorithm also searches for an alternate path extension from that vertex x to v_0 using a DFS. In the example given in Figure 2.4a, the algorithm iterates through the vertices of the path extension E and finds an alternate path extension E' from the neighbour u_1 of v_2. If an alternate path extension is found, a child recursive call is invoked with the updated current path Π, which is $v_0 \rightarrow v_1 \rightarrow v_2$ in our example. Otherwise, if all the vertices in E have already been added to the current path Π, Π is reported as a simple cycle. In our example, the Read-Tarjan algorithm explores both E and E' path extensions, and each leads to the discovery of a cycle.

The Read-Tarjan algorithm also maintains a set of blocked vertices *Blk* for recursion-tree pruning. However, differently from the Johnson algorithm, *Blk* only keeps track of the vertices

Algorithm 5: ReadTarjanBacktrack (Π, $\mathcal{G}$) [RT75]

InOut: Π - the current path
Input: $\mathcal{G}$ - the input graph

1 $v_0 = \Pi$.front(); $v = \Pi$.back(); ▷ The starting vertex and the last added vertex to Π
2 **foreach** $w : \mathcal{N}_{\mathcal{G}}(v)$ **s.t.** *a path extension E from v to* v_0 *exists* **do**
3 $E : w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_k = v_0$; ▷ E can be found using a DFS traversal
4 Blk = $\{\Pi\}$;
5 flag = false;
6 **foreach** $w_i : E$ **do** ▷ Follow the extension E until an alternate path extension is found
7 $\Pi = \Pi$.push(w_i);
8 Blk = Blk $\cup \{w_i\}$;
9 **foreach** $u : \mathcal{N}_{\mathcal{G}}(w_i)$ **s.t.** u.id > v_0.id **do**
10 **if** $u \neq w_{i+1} \wedge u \notin$ Blk **then**
11 flag = DFS(u, v_0, $\mathcal{G}$, Blk) ; ▷ Find an alternate path extension
12 **if** flag = true **then goto** line 13; ▷ If an alternate path extension was found
13 **if** flag = false **then** report cycle Π;
14 **else** ReadTarjanBacktrack(Π, $\mathcal{G}$) ; ▷ Create a child recursive call
15 Delete the vertices appended to Π after v;
16 **Function** DFS *(u, v_0, $\mathcal{G}$,* Blk*)* ▷ Find a simple path from u to v_0 that avoids Blk
Input: u - the current vertex, v_0 - the starting vertex, $\mathcal{G}$ - the input graph
InOut: Blk - blocked vertices
Output: true if a path was found
17 Blk = Blk $\cup \{u\}$;
18 **foreach** $w : \mathcal{N}_{\mathcal{G}}(u)$ **s.t.** w.id > v_0.id **do**
19 **if** $w = v_0$ **then return** true;
20 **else if** $w \notin$ Blk **then**
21 found = DFS(w, v_0, $\mathcal{G}$, Blk);
22 **if** found **then return** true;
23 **return** false;

that cannot lead to new cycles when exploring the current path extension within the same recursive call (see lines 6–12 of Algorithm 5). The vertices in *Blk* are avoided while searching for additional path extensions that branch from the current path extension. For instance, the left subtree of the recursion tree shown in Figure 2.4b demonstrates the exploration of the path extension E shown in Figure 2.4a. During the exploration of E, the vertices $b_1, \ldots, b_k$ are added to *Blk* immediately after visiting w_1, and they are not visited again while exploring E. However, when exploring another path extension E' in the right subtree, the vertices $b_1, \ldots, b_k$ are visited once again (see the dotted path of the right subtree). As a result, the Read-Tarjan algorithm visits $b_1, \ldots, b_k$ twice instead of just once. As we are going to show in Section 4.4, this drawback becomes an advantage when parallelising the Read-Tarjan algorithm because it enables independent exploration of different subtrees of the recursion tree.

3 Fast Enumeration of Maximal Cliques on Manycore Platforms

The Bron-Kerbosch (BK) algorithm with degeneracy ordering [BK73; TTT06; ELS10] is one of the most efficient and widely used algorithms for maximal clique enumeration (MCE) [Con+16; CK08]. As shown in the previous chapter, the BK algorithm is a recursive algorithm that relies heavily on set intersection operations. The data parallelism caused by these set intersection operations can be exploited to accelerate this algorithm. In addition, a large amount of task parallelism is available in the BK algorithm because different regions of the search space can be traversed independently. However, parallelisation of this algorithm on modern computer architectures, such as manycore CPUs, is not straightforward. Because a recursion tree is constructed dynamically and its shape is not known in advance, distributing the work evenly across processing resources poses challenges.

In this chapter, we focus on accelerating the BK algorithm by vectorising set intersection operations that dominate the MCE algorithms and by exploiting task parallelism. First, we prove that the use of hash-join-based set-intersection algorithms within the BK algorithm leads to Pareto-optimal implementations in terms of runtime and memory space compared to those based on merge joins. Then, we present a scalable parallel implementation of the BK algorithm that exploits both data parallelism, by using SIMD-accelerated hash-join-based set intersections, and fine-grained parallelism, by using a shared-memory parallel processing framework that supports dynamic load balancing. Finally, we evaluate our scalable parallel MCE imlementation on different manycore CPUs and demonstrate an order of magnitude speedup compared with a state-of-the-art manycore MCE algorithm.

The rest of this chapter is organised as follows. Section 3.1 gives an overview of our solution and discusses key ideas used for accelerating MCE. Section 3.2 offers a broad complexity analysis of the improved BK algorithm by Eppstein et al. [ELS10], which focusses mainly on the impact of the set-intersection algorithms. A practical implementation of SIMD-accelerated

This chapter is based on the pieces of work published at the *46th Internatonal Conference on Very Large Data Bases (VLDB)*, 2020 [Bla+20a] and at the *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* [Bla+20b].

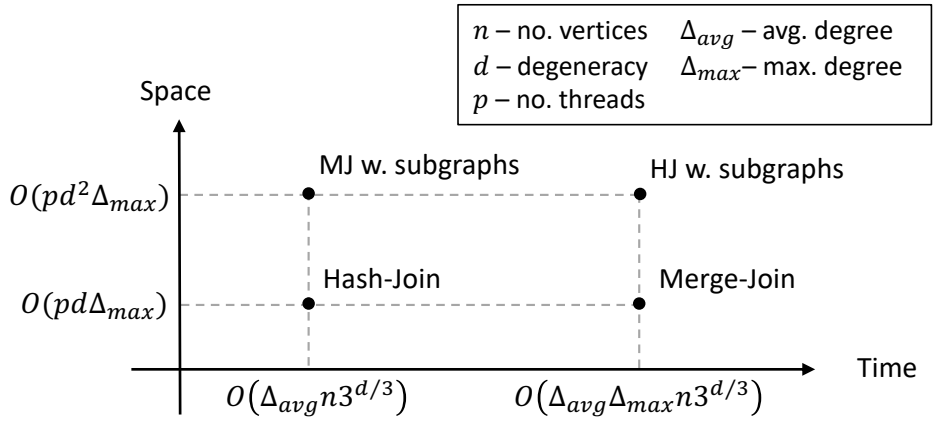


Figure 3.1: Effect of different set-intersection methods on the time and space complexity of the BK algorithm. MJ and HJ stand for merge-join and hash-join, respectively. The solution based on hash joins is Pareto-optimal.

hash-join-based set-intersection implementation is described in Section 3.3. Section 3.4 describes our scalable manycore implementation of the BK algorithm. The experimental evaluation of our solution is given in Section 3.5. The related work is discussed in Section 3.6, and the chapter is concluded in Section 3.7.

3.1 Overview of the Solution

Set intersection operations dominate the execution time of the BK algorithm [HZY18]. These operations are special cases of join operations because sets store only keys and no payloads. When implementing set intersections, one can rely on the two main classes of join algorithms: merge joins and hash joins. Merge joins require both sets to be sorted, while hash joins require at least one of the sets to be hashed. In this chapter, we analyse how intersections based on merge joins and hash joins affect the overall time and space complexity of the BK algorithm.

Intersections in the BK algorithm are performed between the adjacency lists of the input graph and some dynamically-created sets. Because the adjacency lists are static, they can be hashed or sorted in advance. As a result, the complexity of set intersections based on hash joins does not depend on the size of the adjacency lists, which is not the case for set intersections based on merge joins. Considering that the adjacency lists are asymptotically larger than the dynamically created sets, using hash-join-based intersections leads to faster BK implementations. We show that this result is valid both in theory and in practice. Yet, the performance of merge-join-based approaches can be improved using the modified BK algorithm of Eppstein et al. [ELS10]. We show that creating subgraphs at each recursive call, as proposed by Eppstein et al. [ELS10], shrinks the adjacency lists used in the intersections, leading to faster merge-join-based solutions, but at the cost of increased space complexity. Our theoretical analysis results given in Figure 3.1 show that merge-join-based solutions require either more space or more time compared to the hash-join-based solution. The results

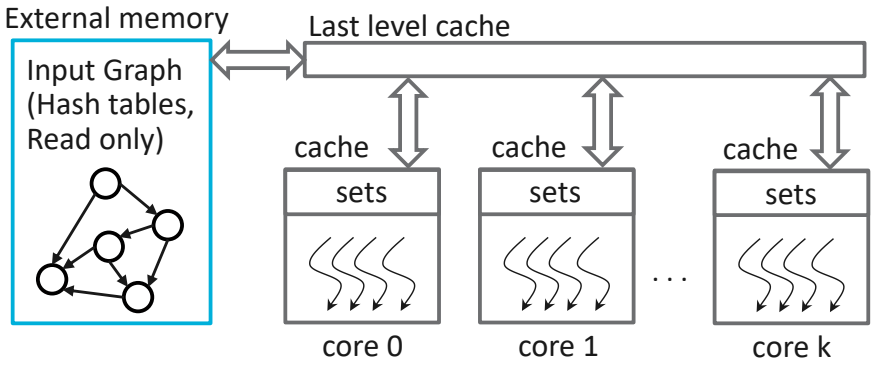


Figure 3.2: Illustration of our manycore setup: each core can execute several concurrent threads and has a private cache in which the sets it creates can reside. The input graph is stored in external memory in the form of hash tables.

given in Figure 3.1 motivate us to focus on BK implementations that use hash-join-based set intersections.

Our manycore implementation exploits both data- and task-level parallelism of the BK algorithm. We scale the algorithm across multiple hardware threads using a framework that utilises dynamic load-balancing, and we exploit data-level parallelism using SIMD-accelerated hash-join-based set intersections. In our manycore setup, illustrated in Figure 3.2, the input graph resides in external memory and the adjacency list of each vertex is stored as a read-only hash table. Hardware threads perform intersections between adjacency lists and local sets, dynamically creating other sets as results. Our intelligent recursion tree exploration approach guarantees that the dynamic memory usage increases only linearly with the number of threads, independently of the number of graph vertices. Thus, it is possible to fit the dynamically-created data structures in the cache space of the CPUs. Lastly, we minimise the task and memory management overheads, which can also constitute a significant part of the execution time. As a result, we are able to run the BK algorithm on a graph with more than 60 million vertices and 1.8 billion edges on a single manycore CPU in only a few minutes. Overall, our implementation is an order of magnitude faster than a state-of-the-art manycore MCE algorithm.

3.2 A Broad Complexity Analysis

This section contributes a broad time and space complexity analysis of the BK algorithm, taking into account both vertex ordering strategies and set intersection algorithms. In particular, we show that whereas hash-join-based set-intersection algorithms lead to ideal complexity bounds, merge-join-based set intersections can lead to a Δ_{max} times higher worst-case time complexity, where Δ_{max} is the maximum node degree of the input graph. We also show that the recursive subgraph-creation scheme given by Eppstein et al. [ELS10] enables the ideal worst-case time complexity to be achieved even when using merge-join-based set intersection

Table 3.1: Worst-case time complexity when using different vertex ordering strategies. SFG stands for scale-free graphs.

Vertex ordering	Time complexity
Arbitrary [TTT06]	$O(3^{n/3})$
Degree [Xu+14]	$O(hn3^{h/3})$
Degeneracy [ELS10]	$O(dn3^{d/3})$
Arbitrary (this work)	$O(n3^{\Delta_{max}/3})$
Arbitrary - SFG (this work)	$O(3^{\Delta_{max}/3})$
Degeneracy (this work)	$O(\Delta_{avg}n3^{d/3})$

algorithms, but the cost of doing so is a d-times higher peak space complexity. These results are summarised in Figure 3.1.

The order in which the vertices are processed when building the recursion tree has a significant impact on the exponential factors of the worst-case complexity. These results are presented in Table 3.1. Tomita et al. [TTT06] assumed an arbitrary ordering of vertices when building the recursion tree and obtained a worst-case complexity of $O(3^{n/3})$. On the other hand, Eppstein et al. [ELS10] proved a worst-case complexity bound of $O(dn3^{d/3})$. Similarly, Xu et al. [Xu+14] focused on the degree-ordering of the vertices, and derived a worst-case complexity bound of $O(hn3^{d/3})$, where h is the h-index of the input graph. Note that in the case of the arbitrary ordering we contribute improved complexity bounds for general and scale-free graphs. In the case of the degeneracy ordering, our results are slightly different from those of Eppstein et al. [ELS10] because we introduce the additional term Δ_{avg}, which is the average node degree of the input graph.

Examples of some real-world graph datasets with their parameters affecting the complexity are given in Table 3.2. These graphs come from the *Network Data Repository* [RA15] and *SNAP* [LK14]. In all of these cases $\Delta_{avg} < d < h < \Delta_{max}$, which means that the algorithms using the degeneracy ordering lead to significantly lower theoretical complexity bounds than the ones with arbitrary and degree ordering. Therefore, we focus on the degeneracy ordering in this work.

3.2.1 Effect of Set-Intersection Algorithms

Because set intersections are heavily used in the Bron-Kerbosch algorithm, it is important to determine the effect of specific set-intersection algorithms on the overall complexity. In this work, we focus on the two most commonly used methods: merge-join- and hash-join-based algorithms.

Let $\rho = |P|$ and $\chi = |X|$ the sizes of the sets P and X in one recursive call of *BKPivot* subroutine of Algorithm 1. Lets assume that a graph has n vertices, m edges, and degeneracy d. The maximum degree of a vertex is Δ_{max}, and the average degree of the vertices is Δ_{avg}. The

Table 3.2: Graph properties. Graphs larger than 1 GB are considered large.

small graph	**abbrv.**	**n**	$\Delta_{\mathbf{avg}}$	**d**	**h**	$\Delta_{\mathbf{max}}$	**size [MB]**
wiki-talk	wt	2.4 M	3.9	131	1055	100 k	64
b-anon	ba	2.9 M	14.3	63	722	4.4 k	80
as-skitter	as	1.7 M	13.1	111	982	35 k	143
livejournal	lj	4.0 M	13.9	213	558	2.6 k	393
topcats	tc	1.8 M	28.4	99	1457	238 k	403
pokec	pk	1.6 M	27.3	47	492	15 k	405
large graph	**abbrv.**	**n**	$\Delta_{\mathbf{avg}}$	**d**	**h**	$\Delta_{\mathbf{max}}$	**size [MB]**
orkut	or	3.1 M	76.3	253	1638	33 k	1740
sinaweibo	sw	59 M	8.9	193	5902	278 k	3891
aff-orkut	ao	8.7 M	74.9	471	6064	318 k	4915
clueweb	cw	148 M	6.1	192	2783	308 k	7373
wiki-link	wl	26 M	41.9	1120	5908	4.2 M	9728
friendster	fr	66 M	54.5	304	2958	5.2 k	30720

following properties hold for ρ, χ, and d [SEF16]:

$$\rho \leq d, \quad \rho + \chi \leq \Delta_{max}, \quad \frac{\Delta_{avg}}{2} \leq d < \Delta_{max}. \tag{3.1}$$

The relationship between the intersection time and the execution time of the *BKPivot* function is given as follows:

Lemma 1. *The complexity of the BKPivot function without its child recursive calls is*

$$O\left((\rho+\chi)\, I\left(\rho,\Delta\right) + \rho\left(I\left(\rho,\Delta\right) + I\left(\chi,\Delta\right)\right)\right), \tag{3.2}$$

where Δ is size of the largest adjacency list accessed, and $I(a,b)$ is the time to intersect a set with a elements and an adjacency list with b elements.

Proof. To determine the pivot vertex, we perform $\rho+\chi$ intersections between the set P and the adjacency list of a vertex from $P \cup X$ (see *getPivot* function), which takes $O\left((\rho+\chi)\, I(\rho,\Delta)\right)$ time. Then, the *BKPivot* function intersects sets P and X up to ρ times with the adjacency list of vertices in $P/N(u)$, which takes $O\left(\rho(I(\rho,\Delta)+I(\chi,\Delta))\right)$ time. The total time is the sum of these two results. □

The next lemma offers the result for the time complexity of the overall algorithm for two significant cases.

Lemma 2. *Let D_0 be the time to execute the BKPivot function without its recursive calls. Then, the time it takes for the BK algorithm with degeneracy ordering to compute all maximal cliques of a graph G is:*

$$D(G) = \begin{cases} O(\Delta_{avg} n 3^{d/3}), & \text{for } D_0 = O(\rho^2 \chi) \\ O(\Delta_{avg}\Delta_{max} n 3^{d/3}), & \text{for } D_0 = O(\rho^2 \chi \Delta_{max}). \end{cases} \tag{3.3}$$

Proof. Using Lemma 5 of Eppstein et al. [ELS10], execution time of the *BKPivot* function including its child recursive calls is $O(\chi 3^{\rho/3})$ when $D_0 = O(\rho^2\chi)$ and $O(\chi\Delta_{max}3^{\rho/3})$ when $D_0 = O(\rho^2\chi\Delta_{max})$. The total cost of all the invocations of *BKPivot* in the *BKDegeneracy* function is

$$\sum_{v} O(\chi 3^{\rho/3}) \leq O(m3^{d/3}) = O(\Delta_{avg} n 3^{d/3}), \tag{3.4}$$

when $D_0 = O(\rho^2\chi)$. Similarly, the total execution time is $O(\Delta_{avg}\Delta_{max}n3^{d/3})$ when $D_0 = O(\rho^2\chi\Delta_{max})$. □

Hash-join-based set intersections require a first set S_a to be hashed. The second set S_b is traversed while performing lookups in S_a. Assuming each lookup takes $O(1)$ time in the worst case, the total time needed for the intersection is $O(|S_b|)$. Constant worst-case lookup time can be achieved using cuckoo hashing [PR01], hopscotch hashing [HST08], and several different perfect hashing algorithms [FKS84; BBD09; BPZ13].

In the hash-join-based BK algorithm, the adjacency list of each vertex is stored in a dedicated hash-table. Given a graph with n vertices and e edges, construction of the hash tables can be achieved in $O(n+e)$ expected time and space complexity, which can be approximated as $O(\Delta_{avg}n)$ when $\Delta_{avg} >= 1$. This pre-processing overhead is significantly lower than the complexity of the BK algorithm.

Theorem 2. *The BK algorithm computes all maximal cliques of a graph $\mathcal{G}$ in $O(\Delta_{avg}n3^{d/3})$ time using degeneracy ordering and hash-join-based set intersections.*

Proof. Given that $I(a,b) = O(a)$ when using hash joins with the second set hashed, the complexity of the *BKPivot* function without its child recursive calls is $O(\rho(\rho+\chi))$ using Lemma 1. The total execution time of the algorithm is obtained by applying Lemma 2 with $D_0 = O(\rho^2\chi)$ given that $O(\rho(\rho+\chi)) \subset O(\rho^2\chi)$. □

Note that Theorem 2 gives a tighter lower bound than $O(dn3^{d/3})$ by Eppstein et al. [ELS10] because $O(\Delta_{avg}) \subset O(d)$. Other hashing algorithms, such as linear probing and double hashing [Knu68], offer weaker bounds on the complexity of hash table lookups. While on average each lookup takes constant time, in the worst-case a lookup can require a linear scan of the hash table. As a result, performing $|S_b|$ lookups in S_a takes $O(|S_a||S_b|)$ time in the worst case, which increases the time complexity of the BK algorithm by Δ_{max} times.

Merge-join-based set intersections require both sets to be sorted. We iterate through both sets in a sequential fashion, looking for common elements. In the worst case, a merge-join-based set intersection performs $O(|S_a|+|S_b|)$ comparisons. In the merge-join-based BK algorithm, sorting the adjacency lists of the input graph is done as a preprocessing step. Because the result of each intersection is also sorted, all the sets remain sorted during the execution of the algorithm without any additional sorting overhead.

Theorem 3. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg}\Delta_{max}n3^{d/3})$ time using degeneracy ordering and merge-join-based set intersections.*

Proof. Because the size of the largest adjacency list accessed can be as large as Δ_{max} and given that $I(a,b) = O(a+b)$ in the merge-join case, the complexity of *BKPivot* without its child recursive calls is $O((\rho+\chi)(\rho+\Delta_{max}))$ using Lemma 1. By applying the Lemma 2 for $D_0 = O(\rho^2\chi\Delta_{max})$ given that $O((\rho+\chi)(\rho+\Delta_{max})) \subset O(\rho^2\chi\Delta_{max})$, we obtain the total execution time of the algorithm. □

In summary, merge-join-based set-intersections lead to a Δ_{max} times higher asymptotic time complexity than hash-join-based set-intersections.

3.2.2 Effect of Recursive Subgraph Creation

Eppstein et al. [ELS10] contributed a subgraph-based BK algorithm, which creates a new subgraph denoted as $H_{P,X}$ before each recursive call. These calls then use their respective subgraphs instead of the original graph G. $H_{P,X}$ is a subgraph of G induced by the vertex set $P \cup X$. However, the edges that exist between the vertices in X in G are not included in $H_{P,X}$. Algorithm 6 shows the *BKSubgraph* function, which replaces the *BKPivot* function of the original BK algorithm, wherein a new function called *createHpxSubgraph* is introduced to create $H_{P,X}$. Note that our formulation of the *BKSubgraph* function given in Algorithm 6 is slightly different from the one given by Eppstein et al. We create only one subgraph per recursive call whereas Eppstein et al.'s formulation [ELS10] creates $O(p)$ subgraphs per call; thus our formulation incurs a lower overhead per call. Note also that the *BKDegeneracy* function has to invoke *BKSubgraph* instead of *BKPivot* when using recursive subgraph creation.

In this section, we evaluate the impact of recursive subgraph creation on the time complexity of the BK algorithm. We evaluate this impact in combination with both merge-join-based and hash-join-based set-intersection algorithms.

Merge join with recursive subgraph creation approach combines the subgraph-based BK algorithm with merge-join-based set intersections. Because the subgraphs shrink with each recursive call, using subgraphs that have smaller adjacency lists than the original graph leads to faster execution of merge-join-based set intersections.

Lemma 3. *createHpxSubgraph can be executed in $O(\rho(\rho+\chi))$ time using merge-join-based set intersections.*

Proof. The *createHpxSubgraph* function iterates through all the vertices in P. For each vertex $u \in P$, it determines the adjacency list $N_{SG}(u)$ of the subgraph SG by intersecting $P \cup X$ with the adjacency list $N_G(u)$ of the original graph G. Because the size of $N_G(u)$ is $O(\rho+\chi)$, the time needed to create $N_{SG}(u)$ for each $u \in P$ using merge-join-based set intersections is $O(\rho+\chi)$.

Algorithm 6: Bron-Kerbosch with subgraph creation

Input: R - set of vertices representing the current clique
P - set of vertices that could form a clique with R
X - set of vertices that can not form a clique with R
$\mathcal{G}$ - the input graph

1 **Function** *BKSubgraph(R, P, X, $\mathcal{G}$)*
2 **if** $P = \emptyset$ **then**
3 **if** $X = \emptyset$ **then** Report R as a maximal clique;
4 **return** ;
5 $\mathcal{SG}$ = createHpxSubgraph(P, X, $\mathcal{G}$);
6 $pivot$ = getPivot($P, X, \mathcal{SG}$); ▷ See lines 10-13 of Algorithm 1
7 **foreach** $v : P \setminus \mathcal{N}_{\mathcal{SG}}(pivot)$ **do**
8 BKSubgraph($R + \{v\}$, $P \cap \mathcal{N}_{\mathcal{SG}}(v)$, $X \cap \mathcal{N}_{\mathcal{SG}}(v)$, $\mathcal{SG}$);
9 $P = P - \{v\}$;
10 $X = X + \{v\}$;
11 **Function** *createHpxSubgraph(P, X, $\mathcal{G}$)*
Output: $H_{P,X}$ subgraph of $\mathcal{G}$
12 **foreach** $u : P$ **do**
13 $\mathcal{N}_{\mathcal{SG}}(u) = (P \cup X) \cap \mathcal{N}_{\mathcal{G}}(u)$;
14 **foreach** $v : \mathcal{N}_{\mathcal{SG}}(u)$ **do**
15 **if** $v \in X$ **then**
16 $\mathcal{N}_{\mathcal{SG}}(v) = \mathcal{N}_{\mathcal{SG}}(v) + \{u\}$;
17 **return** $\mathcal{SG}$

The adjacency lists $N_{SG}(v)$ of $v \in X$ are initially empty. Whenever we find a vertex $v \in X$ in the previously created adjacency list $N_{SG}(u)$, we update $N_{SG}(v)$ by inserting u into it. This insertion can be done in $O(1)$ time by appending the vertex u at the end of $N_{SG}(v)$ because both the adjacency lists and the sets P are always stored in a sorted fashion and traversed as such in merge-join-based implementations. There are $O(\rho + \chi)$ vertices to be examined in $N_{SG}(u)$. Therefore, for each $u \in P$, the time needed to update the adjacency lists $N_{SG}(v)$ of $v \in X$ is $O(\rho + \chi)$. Since p iterations are performed in the outer loop of the *createHpxSubgraph* function, the function executes in $O(\rho(\rho + \chi))$ time. □

Theorem 4. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg} n 3^{d/3})$ time using degeneracy ordering and merge-join-based set intersections in combination with recursive subgraph creation.*

Proof. Since we use the $H_{P,X}$ subgraph instead of the original graph in the *BKSubgraph* function, the size of the largest adjacency list used in that function is $\rho + \chi$. Therefore, the time needed to execute *BKSubgraph* without the *createHpxSubgraph* function can be obtained using Lemma 1, where $\Delta = \rho + \chi$ and $I(a, b) = O(a + b)$, which results in $O(\rho(\rho + \chi))$ time. Re-

calling from Lemma 3 that subgraph creation takes $O(\rho(\rho+\chi))$ time, the total time needed to execute *BKSubgraph* without its recursive calls is $O(\rho(\rho+\chi))$. Similar to the proof of Theorem 2, the overall complexity can be obtained by using Lemma 2 with $D_0 = O(\rho^2\chi)$. □

In summary, recursive subgraph creation enables the BK algorithm based on merge joins to achieve the same time complexity bound as the hash-join-based BK algorithm.

Hash join with recursive subgraph creation method combines hash-join-based set intersections with subgraph-based BK algorithm. Because the hash-join-based implementation operates on hashed adjacency lists, creating a new subgraph requires construction of several new hash tables that store the adjacency lists of the subgraph. Note that the worst-case complexity of constructing a hash table is quadratic in its size given that the worst-case complexity of inserting an element into a hash table is linear in the number of elements for most hashing algorithms.

Theorem 5. *The BK algorithm computes all maximal cliques of a graph G in $O(\Delta_{avg}\Delta_{max}n3^{d/3})$ time using degeneracy ordering and hash-join-based set intersections in combination with recursive subgraph creation.*

Proof. The intersections shown in the line 13 of Algorithm 6 can be performed in $O(\rho+\chi)$ time using the hash-join approach. Therefore, computing all the adjacency lists of a subgraph takes $O(\rho(\rho+\chi))$ time. However, we also have to construct hash tables that store the adjacency lists of the subgraph. Given that a $H_{P,X}$ subgraph has ρ adjacency lists with at most $\rho+\chi$ elements and χ adjacency lists with at most ρ elements, constructing all the hash tables takes $O\left(\rho(\rho+\chi)^2+\chi\rho^2\right) = O\left(\rho(\rho+\chi)^2\right)$ time because of the quadratic complexity of hash table construction. The total time needed to execute *createHpxSubgraph* is $O\left(p(\rho+\chi)^2\right)$.

Using Lemma 1 with $I(a,b) = O(a)$, the execution time of *BKSubgraph* without the *createHpxSubgraph* function is $O(\rho(\rho+\chi))$. Adding it to the time to compute the *createHpxSubgraph* function, the execution time of the *BKSubgraph* function becomes $O\left(\rho(\rho+\chi)^2\right)$. The total execution time of the BK algorithm is obtained by applying Lemma 2 with $D_0 = O(\rho^2\chi\Delta_{max})$ given that χ is at most Δ_{max} and $O(\rho(\rho+\chi)^2) \subset O(\rho^2\chi\Delta_{max})$. □

Theorem 5 shows that recursive subgraph creation increases the worst-case time complexity when using hash-join-based set intersections. However, there exist hashing algorithms that support insertions in constant worst-case time complexity with high probability [DM90; Goo+12; ANS10; ANS09; BE19], leading to a linear worst-case hash table construction complexity. Using such algorithms would reduce the complexity of the BK algorithm with hash-join-based set intersections and recursive subgraph creation to $O(\Delta_{avg}n3^{d/3})$.

3.2.3 Space Complexity

In this section, we perform a space complexity analysis of the BK algorithm assuming the degeneracy ordering of the vertices. In particular, we analyse the impact of recursive subgraph

creation on the peak dynamic memory usage.

At each recursive step, the BK algorithm computes new P, R, and X sets. After that, the BK algorithm invokes a child recursive call and passes the new sets as its parameters. Note that the maximum clique size is upper bounded by $d+1$. Therefore, storing these three sets requires $O(\rho+\chi+d) = O(\Delta_{max})$ space. When the recursion tree is explored in a depth-first-search (DFS) order, the space used for storing the intermediate results is limited to the current execution depth. In addition, when using the degeneracy ordering, the maximum recursion depth is d. Thus, the peak memory consumption of the single-threaded execution of the BK algorithm is $O(d\Delta_{max})$ without taking into account the space needed to store the input graph and the cliques found.

Recursive subgraph creation increases the memory usage further. Each $H_{P,X}$ subgraph uses $O(\rho(\rho+\chi)+\chi\rho) = O(\rho(\rho+\chi)) = O(d\Delta_{max})$ space because it connects either two vertices of P or one from P and one from X [ELS10]. Creating a new subgraph for each recursive call increases the peak memory consumption to $O(d \times d\Delta_{max}) = O(d^2\Delta_{max})$. Therefore, recursive subgraph creation can increase the dynamic memory usage by up to d times. In summary, even though recursive subgraph creation reduces the time complexity of the BK algorithm that uses merge-join-based set intersections, it increases its dynamic memory usage.

The results of our analysis are given in Figure 3.1. Note that when exploring the recursion tree in the DFS order, the dynamic memory usage of the BK algorithm is independent of n. It depends only on the recursion depth d and Δ_{max}.

3.2.4 Parallel Time and Space Complexity

The BK algorithm can be parallelized by considering each recursive call as an independent unit of work (i.e., task). Given N threads of execution, N DFS-based explorations of the recursion tree can be performed concurrently. Based on Brent's theorem, we have $T_N \leq T_1/N + T_\infty$, where T_1 is the execution time using a single core, T_∞ is the execution time using infinitely many cores, and T_N is the execution time using N cores [Bre74]. In the case of the BK algorithm with degeneracy ordering, $T_1(n) = O(\Delta_{avg} n 3^{d/3})$ and $T_\infty(n) = O(d)$ because d is the maximum depth of the recursion tree. Thus, the execution time using p threads of execution is

$$T_p(n) = O\left(\frac{\Delta_{avg} n}{p} 3^{d/3} + d\right). \tag{3.5}$$

As a result, we expect the performance of parallel implementations that take advantage of dynamic task scheduling frameworks [Qui04; Blu+96b; Kuk07] to scale linearly with the number of threads (strong scaling). In addition, when d is constant and the number of hardware threads (p) is a linear function of n, the worst-case time complexity is also a constant. Hence, a weak performance scaling can be achieved as well.

When we execute the BK algorithm on p threads, we explore N different DFS paths of the

recursion tree in parallel. Because the memory consumption of each thread is equal to the memory usage of a single DFS path, the space needed for executing the parallel algorithm is up to p times larger than the single-threaded results given in Section 3.2.3. We conclude that the peak memory consumption is $O(Nd\Delta_{max})$ without subgraphs while it is $O(Nd^2\Delta_{max})$ with subgraphs. Considering both hash-join-based and merge-join-based set intersections, we summarise these results in Figure 3.1.

3.2.5 Arbitrary Vertex Orderings

In this section, we derive new complexity bounds for arbitrary vertex orderings. When using arbitrary vertex orderings, the size of set P is no longer upper bounded by d but by the maximum vertex degree Δ_{max}. Based on this observation, the time complexity of the BK algorithm that uses arbitrary vertex ordering is $O(n3^{\Delta_{max}/3})$, because each invocation of the *BKPivot* function takes $O(3^{\Delta_{max}/3})$ time.

A better complexity bound can be derived for scale-free graphs. In scale-free graphs, the probability of a vertex having degree Δ is $\mathcal{P}(\Delta) \sim \Delta^{-\gamma}$, where the γ parameter is typically between 2 and 3 [BP16]. Many real-world graphs are scale free, such as the World Wide Web, protein-interaction, and email networks. These graphs have a limited number of highly-connected vertices. When the BK algorithm is executed on scale-free graphs, most of its execution time is spent when starting from those highly-connected vertices.

Theorem 6. *The BK algorithm computes all maximal cliques of a scale-free graph in $O(3^{\Delta_{max}/3})$ time using an arbitrary ordering of vertices when $\Delta_{max}^{\gamma} \leq 3^{(\Delta_{max}-1)/3}$.*

Proof. Each invocation of *BKPivot* in the *BKDegeneracy* function takes $O(3^{\Delta/3})$ time, for an arbitrary vertex ordering. The sum of the cost of all invocations is

$$\sum_{v} O(3^{\Delta/3}) = O\left(\sum_{\Delta=1}^{\Delta_{max}} N(\Delta)3^{\Delta/3}\right), \tag{3.6}$$

where $N(\Delta)$ is the number of vertices with degree Δ. In scale-free graphs, the degrees follow a power-law distribution, i.e., the number of vertices with degree Δ is proportional to $n\Delta^{-\gamma}$. For scale-free graphs, it also holds that $n = O\left(\Delta_{max}^{\gamma-1}\right)$ [BP16]. By using these properties, we obtain

$$\begin{aligned} O\left(\sum_{\Delta=1}^{\Delta_{max}} N(\Delta)3^{\Delta/3}\right) &= O\left(n\sum_{\Delta=1}^{\Delta_{max}} \frac{3^{\Delta/3}}{\Delta^{\gamma}}\right) \\ &\leq O\left(n\frac{3^{\Delta_{max}/3}}{\Delta_{max}^{\gamma-1}}\right) \leq O\left(3^{\Delta_{max}/3}\right), \end{aligned} \tag{3.7}$$

where $3^{\Delta/3}/\Delta^{\gamma} \leq 3^{\Delta_{max}/3}/\Delta_{max}^{\gamma}$ holds for every Δ_{max} if $3^{(\Delta_{max}-1)/3} \geq \Delta_{max}^{\gamma}$. □

In real-world graphs, the condition $3^{(\Delta_{max}-1)/3} \geq \Delta_{max}^{\gamma}$ almost always holds. For example, when $\gamma = 3$, the maximum node degree (i.e., Δ_{max}) needs to be larger than 29.

Theorem 6 shows that the worst-case complexity of the BK algorithm using an arbitrary vertex ordering depends only on the time to process the vertex with the maximum degree. Note that Δ_{max} is much smaller than n in real-world graphs even though $\Delta_{max} = O(n)$ (see Table 3.2). Effectively, we have derived a new bound that is significantly tighter than the $O(3^{n/3})$ bound reported in Tomita et al. [TTT06].

3.3 Vectorized Set Intersections

Set intersection operations are the dominant part of the BK algorithm [HZY18]. Performing set intersections is required both when determining the pivot vertex (line 12 in Algorithm 1) and when constructing the new sets P and X (line 7 in Algorithm 1). To improve performance, it is crucial to reduce the time spent on these operations. In this section, we describe our implementations of SIMD-accelerated set intersection. We show that our SIMD-accelerated hash-join-based set intersection implementation outperforms state-of-the-art methods when the sets involved in intersections have disproportionate sizes, which frequently occurs when executing the BK algorithm.

3.3.1 Merge-Join-Based Set Intersection

Our SIMD-accelerated intersection implementation based on merge joins uses a cache-friendly data structure called the cache-aligned list (*CAlist*), which reduces cache misses and increases processing rates by using vector instructions. The sets are stored in lists of cache-aligned buckets, where the size of each bucket is divisible by the L2 cache line size and stores several vertex IDs as well as a pointer to the next bucket. Set intersections are parallelised using vector instructions, which enable performing several comparisons in only a few clock cycles. An intersection between two sets is executed by iterating through the smaller set and checking whether the vertices of the smaller set also exist in the larger set. Figure 3.3 illustrates the process of merge-join-based set intersection. Whether the vertex 3 of the set A exists in the first bucket of the set B can be determined using two SIMD instructions. The first SIMD instruction replicates the vertex ID in a vector, whose size is the same as the number of elements in the bucket. The second SIMD instruction compares this vector with the contents of the bucket. As a result, we obtain a bit vector that indicates the position of the vertex 3 within the current bucket of the set B. If the resulting bit vector is nonzero, the vertex 3 belongs to both sets. In this particular example, we see that the next element of the set A (i.e., vertex 5) cannot exist in the first bucket of the set B because it has a value greater than the value of the last element of the bucket. Therefore, we can simply skip to the next bucket of the set B and repeat the same process for the next vertex 5 of the set A.

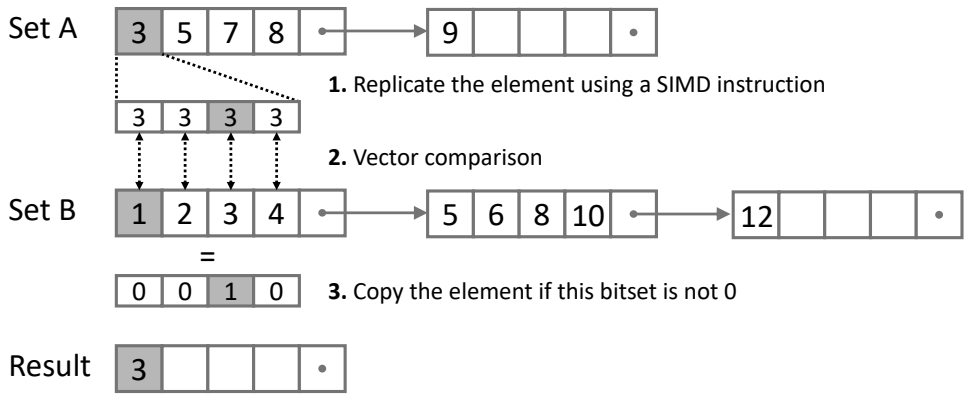


Figure 3.3: Data-parallel set intersections using CAlist. Sets are represented as linked lists of buckets containing several element. Two SIMD instructions are used to verify whether an element of one set belongs to the bucked of the other set.

3.3.2 Hash-Join-Based Set Intersection

We have developed a hash-join-based set intersection algorithm, called *SimpleHashSet*, which constructs hopscotch hash tables [HST08] and performs SIMD-accelerated table lookups. We use hopscotch hash tables to achieve $O(1)$ worst-case complexity for the lookups. Even though the construction of hopscotch hash tables differs from the construction of the tables used by linear probing implementations, the table lookups can be performed in exactly the same way. Our SIMD implementation of table lookups is based on the SIMD-accelerated linear probing implementation of Polychroniou et al. [PRR15]. However, we support only unique integer keys without payloads. Thus, our design is much simpler than that of Polychroniou et al. [PRR15]. Note that the set difference operations used by the BK algorithm can also be implemented using our SIMD-accelerated hash-table lookups.

We build a dedicated hopscotch hash table for each vertex of the input graph to store its adjacency list. Hopscotch hash tables are constructed in such a way that each key is found within H entries of the address computed by the hash function [HST08]. In our implementation, H is the number of integer keys that fit into one cache line. When computing the size of a hash table, we multiply the size of the respective adjacency list by two and round it up to the nearest power of two. Note that the hopscotch hash tables require a hash function from a universal family. We take advantage of multiplicative universal hashing, which uses one multiplication, one addition, and one bit-shift operation [Woe99; Die+97].

3.3.3 Comparison to Other Algorithms

We perform an experimental study of set intersection algorithms *(i)* to show the efficiency of our algorithms in comparison to prior solutions and *(ii)* to understand when hash joins should be preferred over merge joins. Our experiments in this section are performed on an Intel[I] Xeon

[I]Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Table 3.3: Hardware platforms used for for the experimental evaluations in Chapter 3.

platform	Intel KNL [Sod15]	Intel Xeon Skylake
no. cores	64	48
no. threads	256	96
SIMD instr.	AVX-512	AVX-512
memory	110 GB	360 GB
L1d cache	32 KB per core	32 KB per core
L2 cache	1 MB per 2 cores	1 MB per core
L3 cache	none	38.5 MB

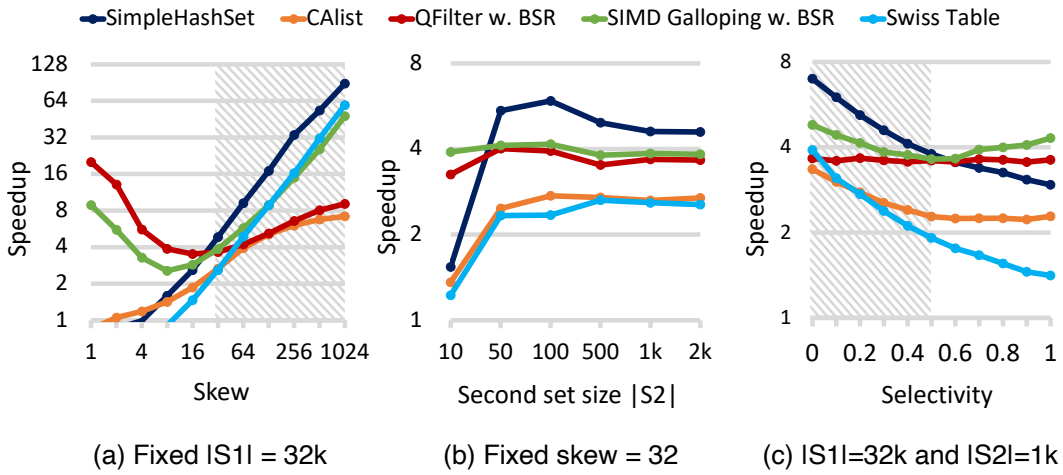


Figure 3.4: Speedup of SIMD-accelerated set intersection algorithms compared to scalar-merge-based set intersections. The shaded regions represent the cases that often occur in the BK algorithm. Our *SimpleHashSet* is faster than the other solutions in these shaded regions.

Phi 7210 - Knights Landing (KNL) processor [Sod15], described in Table 3.3. Intel KNL supports the state-of-the-art AVX-512 instructions, where each vector register enables operations on sixteen 32-bit integer operands in parallel. The following state-of-the-art implementations of set intersection algorithms that use SIMD instructions were used in the comparison.

QFilter is a merge-join-based set intersection algorithm optimised for graph processing by Han et al. [HZY18]. It uses a compressed bit-vector representation of graph vertices, called BSR, and it accelerates the set-intersection operations using 128-bit vector registers and the AVX2 instruction set. The main drawback of this method is that it requires a time-consuming reordering of the input graph vertices in order to achieve high-performance intersections.

Galloping is a merge-join-based intersection algorithm that locates the members of the first set in the second set using binary search, and it can be accelerated using SIMD instructions [LBK16]. Han et al. [HZY18] Optimised this approach to use their compressed bit-vector representation. We refer to this implementation as *SIMD Galloping with BSR*.

CAlist is our SIMD-accelerated merge-join-based set-intersection implementation described in Section 3.3.1. Because we use AVX-512 instructions, the size of each bucket of the cache-

aligned linked list is equal to 512 bits. This approach can be easily adapted to take advantage of even wider vector registers by increasing the bucket sizes.

Swiss Table is a highly-optimized open-source hash table library by Abseil adapted from Google's C++ codebase [Abs17].

We use a microbenchmarking approach in which we randomly generate and intersect two sets $S1$ and $S2$ and vary their sizes. Within the BK algorithm, $S1$ is an adjacency list of the graph, which is hashed in the hash-join-based case, and $S2$ is either the set P or the set X. Figure 3.4 shows the speedup achieved by different set intersection strategies over a scalar merge join strategy that is used as the baseline. We fix *density* of the sets (i.e., the ratio between the larger set size and the range of elements in sets [HZY18]) to 0.5 in the case of QFilter and SIMD Galloping with BSR, and to 0.1 in all other cases. The reason for using the higher density in QFilter and SIMD Galloping with the BSR cases is that they benefit from graph reordering [HZY18], which increases the densities of the sets and enables more efficient intersections.

Figure 3.4a compares the performance of different set intersection implementations when the ratio between the set sizes (i.e., *skew*) varies. We fix *selectivity* of the intersections (i.e., the ratio between the size of the result and the smaller set) to 0.3 because the BK algorithm using scalar-merge-based set intersections spends 50 – 60% of its set intersection time on set intersections with a selectivity lower than 0.3 on average across all the graphs from Table 3.2. Then, we fix the size of the set $S1$ to 32000, which is in the order of the average size of the set $S1$ observed when processing the large graphs. Note that this average is much larger than Δ_{avg} because the vertices with higher degrees participate in intersections much more frequently. We then vary the skew between 1 and 1024. We see that *SimpleHashSet* is preferable when the set sizes are disproportionate (the shaded region) whereas QFilter is preferable when the skew is small. (i.e., when $S1$ and $S2$ are similar in size).

When operating on large graphs, the BK algorithm that uses scalar-merge-based set intersections spends more than 80% of its intersection time on intersecting sets with a skew larger than 32. Thus, in Figure 3.4b, we fix the skew to $|S1|/|S2| = 32$ while keeping the selectivity at 0.3 and we vary the size of both sets proportionally until $|S1| = 64000$. It is clear that *SimpleHashSet* outperforms all other set intersections when $|S2|$ is not extremely small. In Figure 3.4c, we keep the skew at $|S1|/|S2| = 32$, fix the size of $S1$ to 32000, and vary the selectivity. Averaged across the large graphs from Table 3.2, the BK algorithm that uses the scalar merge spends almost 70% of its intersection time executing the intersections with a selectivity lower than 0.5. Our *SimpleHashSet* is faster than the other set intersection algorithms exactly in that region.

In conclusion, hash-join-based set intersections are preferable to merge-join-based ones when the set sizes involved in the intersections are highly skewed, which is often the case for the BK algorithm. Furthermore, our *SimpleHashSet* method is competitive against state-of-the-art set intersection methods such as QFilter without requiring compressed bit-vector representations such as BSR.

3.4 Manycore Implementation

In this section, we build on the theoretical results of Section 3.2 and develop a shared-memory parallel implementation of the BK algorithm that exploits task-level parallelism and guarantees a worst-case complexity of $O(pd\Delta_{max})$ on the peak dynamic memory consumption. The pseudocode of our final parallel algorithm for maximal clique enumeration based on the BK algorithm is given in Algorithm 7. In this pseudocode, the *Vector_HJ_Intersection* function represents our vectorised hash-join-based set intersection implementation described in the previous section. First, we describe our initial parallel implementation of the BK algorithm that exploits task-level parallelism. Next, we discuss the optimisations that minimise the dynamic memory usage and maximise the scalability of our software implementation.

The experiments presented in this section are performed on the Intel KNL platform (see Table 3.3) using 256 hardware threads. We use the Intel VTune Amplifier version 2018.3 to obtain the execution time breakdown. Tools such as Intel VTune and Valgrind Massif [Sew08] can be used to perform dynamic memory usage profiling. However, they are extremely slow in doing so. Therefore, we extract the dynamic memory usage profile by instrumenting our code. We keep track of the dynamic memory allocations and deallocations performed by individual threads on the relevant data structures, and periodically sample their sum. A new sample is collected when a certain number of memory allocations have been performed, which is tracked by an atomic counter.

3.4.1 Fine-Grained Parallelisation

As already explained in Section 1.2, simply parallelising the BK algorithm by executing its outer loop shown in line 2 of Algorithm 2 in parallel does not result in a scalable algorithm. Because iterations of this outer loop execute recursion trees of different sizes, each recursion tree should be executed using several threads to achieve scalable execution of the BK algorithm. Furthermore, to keep memory usage minimal, each thread should explore a portion of a recursion tree in a depth-first manner, as shown in Figure 2.2. For this purpose, we use the Intel *Threading Building Blocks* (TBB) software framework [Kuk07], which enables the decomposition of a recursion tree into tasks that can be independently executed by the available worker threads (see Section 2.2). Thus, several threads can explore the same recursion tree. In addition, TBB implements a dynamic scheduler that can dispatch tasks to parallel worker threads, where the load balance between these threads is achieved using the work-stealing approach [BL99]. This scheduler forces each worker thread to execute the tasks it generates in a depth-first fashion. Therefore, TBB enables exploring a recursion tree of the BK algorithm in a parallel depth-first manner.

Our initial manycore implementation wraps each recursive call to the *BKPivot* function of Algorithm 1 in a task. In each iteration of its `foreach` loop, memory for the new sets $P' = P \cap \mathcal{N}_G(v)$ and $X' = X \cap \mathcal{N}_G(v)$ is allocated, and a new task is *spawned* with these sets as parameters. We refer to the original task as the parent task, and to the spawned tasks as

Algorithm 7: Our parallel BK algorithm

Input: R - set of vertices representing the current clique
P - candidate vertex set
X - exclude vertex set
$\mathcal{G}$ - the input graph

```
1  Task BKTask (R, P, X, G)
2      if |R| = 1 then                          ▷ Delay the creation of initial P and X sets
3          u = R.back() ;                       ▷ u is the only element of R
4          I_u = The index of u in V;           ▷ V is ordered in degeneracy ordering
5          foreach w : N_G(u) do
6              I_w = The index of w in V;       ▷ V is ordered in degeneracy ordering
7              if I_w > I_u then P = P + {w} ;  ▷ P and X sets are initially empty
8              else X = X + {w} ;
9      if P = ∅ then
10         if X = ∅ then Report R as a maximal clique;
11         return ;
12     foreach v : P ∪ X do                     ▷ Find pivot vertex
13         t_v = |Vector_HJ_Intersection(P, N_G(v))| ;   ▷ Returns |P ∩ N_G(v)|
14     pivot = arg max_v(t_v);
15     foreach v : P \ N_G(pivot) do
16         P' = Vector_HJ_Intersection(P, N_G(v));      ▷ Returns P ∩ N_G(v)
17         X' = Vector_HJ_Intersection(X, N_G(v));      ▷ Returns X ∩ N_G(v)
18         if |P| + |X| > t_t then              ▷ Task grouping
19             spawn BKTask(R + {v}, P', X', G);        ▷ Create new task
20         else
21             BKTask(R + {v}, P', X', G);      ▷ Execute this task as a function
22         P = P − {v};
23         X = X + {v};
24     sync;                                    ▷ Wait for all spawned tasks
25 Function ParallelBK (G(V, E))
26     Order vertices V in G using degeneracy ordering;
27     Hash the adjacency lists of G;
28     parallel foreach v_i : V do
29         spawn BKPivot({v_i}, ∅, ∅, G);
30     sync;                                    ▷ Wait for all spawned tasks
```

child tasks. When the `foreach` loop completes, we wait for all the child tasks to complete before destroying the parent task. Note that the *BKDegeneracy* function, which is the root of the recursion tree, is also implemented as a task, and the iterations of its `foreach` loop are executed in parallel using TBB's `parallel_for` loop construct. When recursive subgraph creation is enabled, each task creates a $H_{P,X}$ subgraph from the P and X sets as described in Algorithm 6. Parallel DFS exploration of the recursion tree is enforced by the TBB scheduler by

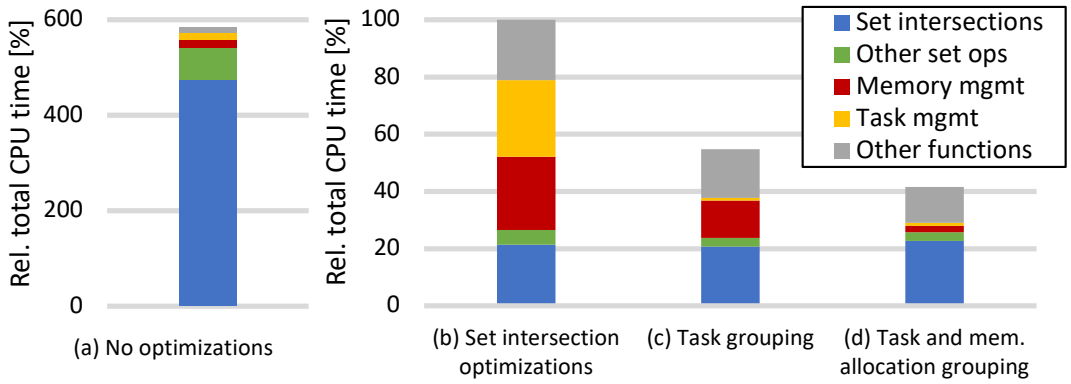


Figure 3.5: Impact of various optimizations on the total CPU time used by the manycore BK implementation when processing the *orkut* graph. In (a), we use scalar-merge-based intersections, while (b), (c), and (d) we use our *SimpleHashSet*. The use of our vectorised hash-join-based set intersection reduces the execution time 6×. Task and memory management overheads are practically non-existent after our manycore optimisations.

default. Each worker thread simply prioritises the task that it spawned most recently.

Figure 3.5 shows the execution time breakdown of our manycore BK implementation. In the unoptimized case shown in Figure 3.5a, set intersections based on the scalar merge approach dominate the execution time. However, using our *SimpleHashSet*, described in Section 3.3, accelerates the set intersections by 22×, as shown in Figure 3.5b. This result matches the results of Figure 3.4a, where *SimpleHashSet* is up to 128 times faster than the scalar merge approach for the cases that frequently occur in the BK algorithm. However, once the set intersection implementation is optimised, the task and memory management overheads can constitute up to 50% of the total CPU time, as shown in Figure 3.5b. In addition, our initial implementation does not achieve the ideal space complexity results reported in Section 3.2.4, and uses more memory than necessary. In this section, we discuss these problems in detail and offer our solutions.

3.4.2 Minimising Dynamic Memory Usage

The initial manycore implementation of the BK algorithm described in the previous section uses more dynamic memory than what is predicted in Section 3.2.4. The main reason is that a task executing on a TBB thread spawns all its child tasks and allocates memory for them while it is still executing. After completion of the current task, the thread can switch to one of these child tasks. However, the remaining child tasks occupy memory that is not yet being used, causing two main problems: 1) Dynamic memory usage depends on the number of vertices n. 2) The memory used by the *BKPivot* task shown in Algorithm 1 can be up to d times larger than necessary.

The first problem is caused by the `parallel_for` that implements the `foreach` loop of the *BKDegeneracy* function. The default behaviour of TBB's `parallel_for` loop is to heuristically

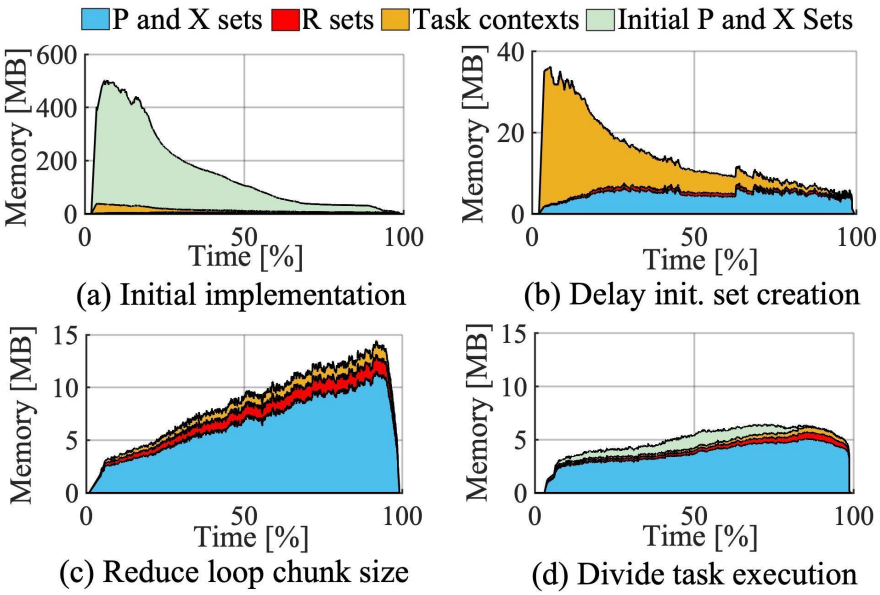


Figure 3.6: Dynamic memory usage over time for the *orkut* graph. Our optimizations reduce the peak dynamic memory usage by 80× while affecting the runtime only marginally.

group its iterations into *chunks*, and then a worker thread sequentially executes an entire chunk before starting to work on another chunk [VAR19]. Each iteration of the main loop of the *BKDegeneracy* function spawns a task with an initial pair of P and X sets (see Algorithm 2). By default, TBB does not limit the chunk size, so the `parallel_for` loop might have a chunk made of up to n loop iterations. A thread executing that chunk would then allocate $O(n(\Delta_{max}+C))$ memory, assuming that C is the memory used by the task context in addition to the sets. The resulting memory allocations could easily become the dominant component of dynamic memory usage when processing large graphs. Figure 3.6a shows that the initial sets consume a significant amount of memory in the case of the *orkut* graph. One way to solve this problem is to delay the creation of the initial sets until the start of the corresponding *BKPivot* function, which ensures that each thread creates at most one pair of the initial sets. This solution is shown in lines 2-8 of Algorithm 7. By doing so, we reduce the dynamic memory usage by 15× for the *orkut* graph, compared to the initial implementation (Figure 3.6b). However, the task contexts created in the `parallel_for` loop of the *BKDegeneracy* function still represent a large part of the execution time. This problem can be solved by limiting the chunk size to one, which results in each chunk creating exactly one task. As Figure 3.6c shows, this optimisation further reduces the dynamic memory usage to less than half. Note that now the space complexity no longer depends on n.

The second problem occurs because each *BKPivot* task spawns $O(|P|) \subset O(d)$ child tasks after creating the respective P and X sets, which leads to usage of $O(d)$-fold more dynamic memory than necessary within recursive calls. Then, the space complexity becomes d times higher than what is predicted in Section 3.2.4. To solve this issue, we enable different threads to execute different iterations of the `foreach` loop shown in line 6 of Algorithm 1 (see Figure 3.7).

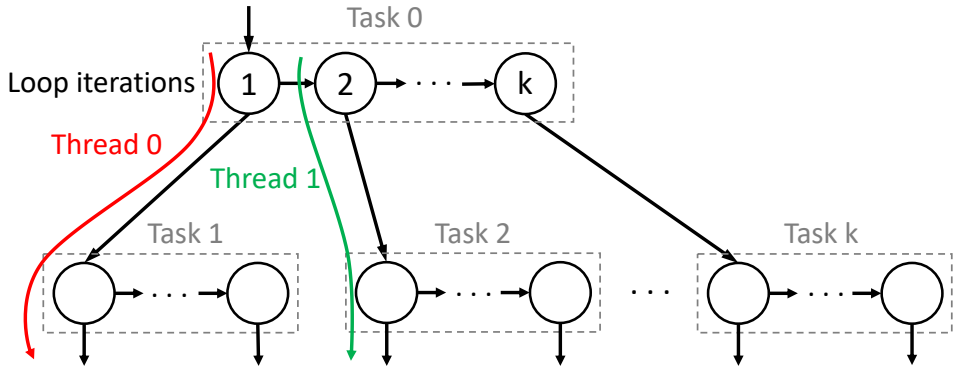


Figure 3.7: Reducing dynamic memory usage by enabling different threads to execute different loop iterations (circles). Thread 0 starts executing Task 1 immediately after the first loop iteration of Task 0 instead of executing the entire Task 0 before Task 1. The creation of temporary data structures in other loop iterations of Task 0 is delayed, which reduces dynamic memory usage.

After executing a loop iteration, the worker thread can either switch to the next loop iteration or to its child task. Using TBB's *scheduler bypass* feature, we force the thread to switch to the child task (Thread 0 of Figure 3.7) and return the context of the parent task to the scheduler. Another thread can later continue executing the parent task by picking up its context from the scheduler (Thread 1 of Figure 3.7). This approach is similar to continuation stealing introduced by *Cilk-5* [FLR98; Lee+10]. By executing the child task before the next loop iteration, each worker thread spawns only one task at a time instead of spawning $O(d)$ tasks at once. Figure 3.6d shows that this optimisation further reduces dynamic memory usage by 50% when processing the *orkut* graph.

As a result of the previous optimisations, our manycore implementation of the BK algorithm uses $O(pd\Delta_{max})$ space for dynamic memory, as we predict in Section 3.2.4. Dynamic memory usage is reduced by 80× in the case of the *orkut* graph (see Figure 3.6), and from 30× to 180× when processing small graphs. The highest dynamic memory usage measured when processing these seven graphs is around 10 MB, which is significantly lower than the cumulative cache capacity of the Intel KNL processors (see Table 3.3).

3.4.3 Task Grouping

If more time is spent on managing tasks rather than executing them, the manycore implementation will not scale well. As Figure 3.5b suggests, more than 25% of the time is spent on task management. One way of reducing the task management overheads is to group several recursive calls in a single task. However, grouping too many calls in a task can lead to load imbalances, which increases the idle time of the worker threads. In this section, we describe a task grouping heuristic that aims to marginalise the impact of task management on the runtime without causing resource underutilisation. This goal is fulfilled by creating sufficiently

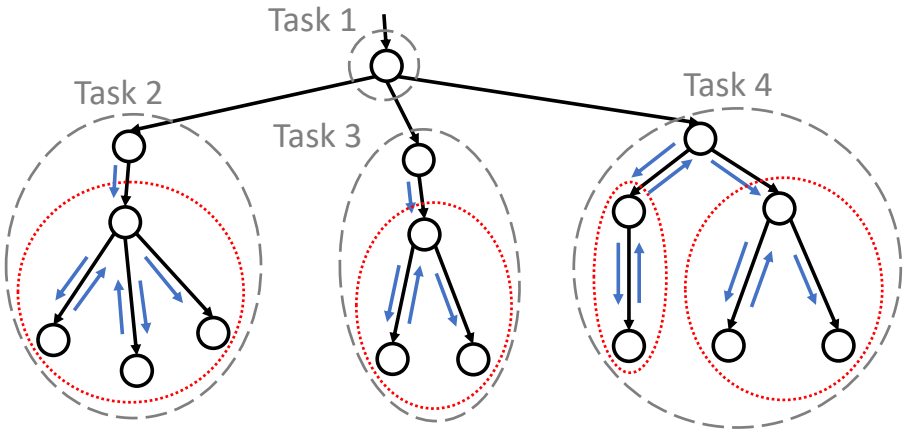


Figure 3.8: Manycore optimisations: task grouping and memory allocation grouping. Circles represent recursive calls, dashed ellipses the tasks, and dotted ellipses the recursive calls that share the same pre-allocated memory region.

complex tasks.

Theorem 2 shows that the complexity of executing a recursive call without its child recursive calls depends on the size of the sets P and X as $O(|P|(|P|+|X|))$. Both sets typically become smaller as we move deeper in the recursion tree. Therefore, we heuristically restrict task grouping only to the recursive calls near the bottom of the recursion tree, as it is shown in line 18 of Algorithm 7. We create the task groups implicitly by not spawning new tasks if the cardinality of the corresponding $P \cup X$ set is smaller than a task threshold t_t, and execute the following recursive call sequentially instead, as depicted in Figure 3.8. Figure 3.5c shows that the task management overheads become negligible after the optimisation. In addition, the memory management overheads are also indirectly reduced. We will study the choice of the empirical parameter t_t in Section 3.4.5 and show that it is not particularly critical.

This optimisation has no side effect on the peak space complexity because the recursive calls within a task are implicitly executed in DFS order by our C++ implementation. Thus, the peak space complexity bounds derived for DFS-based processing in Section 3.2.4 apply without any changes.

3.4.4 Memory Allocation Grouping

Frequent memory allocations and deallocations by multiple threads can cause contention, lead to performance overheads, and limit scalability. TBB's scalable memory allocator alleviates such problems, but it cannot eliminate them completely. The main reason is the frequent dynamic allocations and deallocations of the P and X sets. Given that the majority of the recursive calls are short-lived, especially towards the leaves of the recursion tree, memory allocation can easily become a scalability bottleneck. Figure 3.5c shows that, after task grouping, memory management overheads still take up to 25% of the total CPU time.

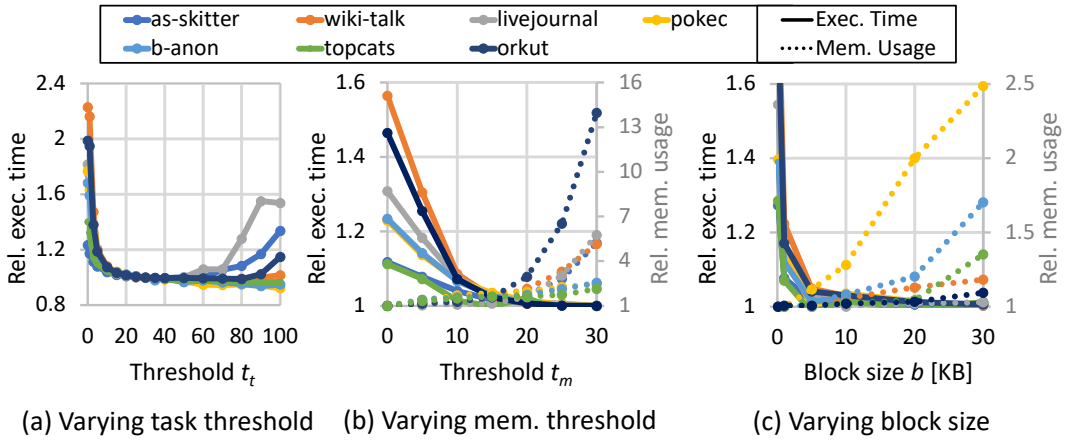


Figure 3.9: Sensitivity of our manycore implementation to the parameters t_t, t_m and b. Graph (a) shows the execution time relative to $t_t = 30$. Graph (b) shows the execution time relative to $t_m = 30$ and the peak memory usage relative to $t_m = 0$. Graph (c) shows the execution time relative to $b = 30$ KB and the peak memory usage relative to $b = 100$ B.

We introduce a memory allocation grouping method to reduce the memory management overheads of our implementation. To reduce the number of memory allocations, we create a large block of memory, in which the sets created by several consecutive recursive calls are placed. Considering that the sets become smaller and the memory allocations become more frequent as we move deeper in the recursion tree, grouping the memory allocations that originate near the bottom of the tree is a necessity. Similarly to our task grouping approach, we introduce a memory threshold t_m and group memory allocations of a recursive call and all its child calls when $|P| + |X| \leq t_m$. We constrain t_m to be smaller than the task threshold t_t to ensure that memory blocks are not shared by different threads and that there is no need to synchronise the accesses to these blocks.

We denote the size of a pre-allocated memory block as b. When b is not large enough to accommodate all the sets, a singly-linked list of such blocks is created. Using a too large b increases the memory usage whereas using a too small one increases the number of allocations and negatively impacts the performance. Figure 3.5d shows that our memory allocation grouping method using $t_m = 20$ and $b = 20$ KB virtually eliminates the memory management overheads.

3.4.5 Sensitivity Analysis

This section evaluates the impact of the manycore implementation parameters t_t, t_m, and b on execution time and memory usage. Experiments are performed on Intel KNL using 256 hardware threads and cover all small graphs and one large graph (i.e., *orkut*) from Table 3.2.

First, we evaluate the task-grouping optimisation in isolation. Figure 3.9a shows the impact of the task threshold t_t on the execution time. In all the cases evaluated, the best performance is

achieved when t_t is between 20 and 50. Note that t_t does not influence the dynamic memory usage. Next, we set $t_t = 30$ and evaluate the memory-allocation-grouping optimisation. In Figure 3.9b, we set $b = 20$ KB and vary the memory threshold t_m. Across all the graphs tested, the lowest execution time and dynamic memory usage combinations are achieved when the range of t_m is between 10 and 20. In fact, the highest performance is achieved when $t_m = 30$. However, its dynamic memory usage can be up to 13× higher than that of the baseline (i.e., $t_m = 0$), which disables the memory grouping optimisation. A good trade-off is achieved when $t_m = 20$. In this case, the execution time is within 5% of the optimal and the memory usage is at most 3× higher than that of the baseline. Finally, in Figure 3.9c we set $t_m = 20$ and vary the block size b. The execution times decrease as we increase b and reach their optimal values at $b = 20$ KB. Note that the dynamic memory usage increases linearly with the block size after a certain point. When $b = 20$ KB, the memory usage can be up to 2× higher than that of the baseline that does not use memory preallocation (i.e., $b = 0$), which is not a significant overhead.

Thus, our experiments suggest that none of the empirical parameters is particularly critical: *i)* the results are consistent across all the graphs we use and *ii)* for each parameter there exists a reasonable range where the optimisations are similarly effective as with the very best values. Therefore, we set $t_t = 30$ for task grouping and $b = 20$ KB, $t_m = 20$ for memory allocation grouping. We use these values in all the experiments performed in the remainder of this chapter.

3.5 Experimental Results

In this section, we first show the impact of our algorithmic and implementation choices as well as our optimisations on execution time, memory consumption, and manycore scalability. Then, we compare our optimised implementation with state-of-the-art references in terms of both single-threaded and multi-threaded performance.

3.5.1 Experimental Setup

In the experiments, we use two platforms: Intel KNL and Intel Xeon Skylake. We developed our code on Intel KNL and ran most of the analyses there; yet, for completeness, we ran the scalability analysis and the comparisons to competing implementations also on Intel Xeon Skylake processors available in Google Cloud's Compute Engine. The properties of these platforms are summarised in Table 3.3. We build our code using GCC v8.3.1 with the -O3 optimisation flag. To exploit task parallelism, we use the version 2019_U9 of the Intel TBB framework.

As already mentioned, the graph datasets that we use are obtained from the *Network Data Repository* [RA15] and *SNAP* [LK14] (see Table 3.2). The large graphs from the table represent a subset of massive network data from the *Network Data Repository*. In our experiments, the

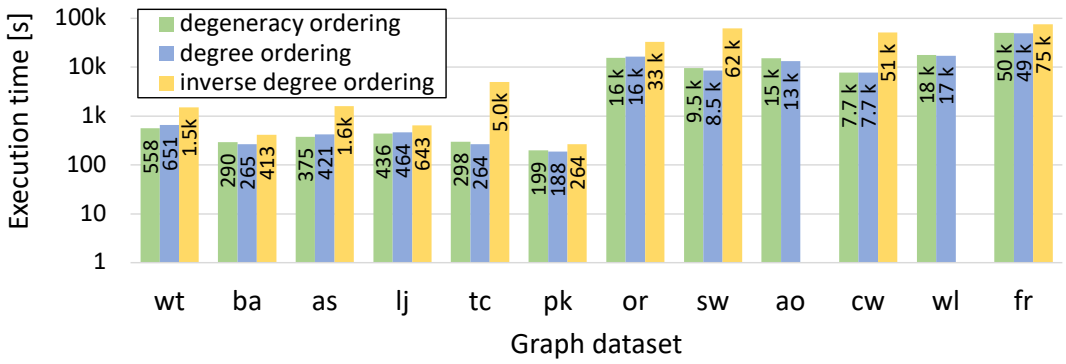


Figure 3.10: Impact of various vertex ordering techniques on the single-threaded performance of the BK algorithm when using Intel KNL. The experiments using the inverse degree ordering did not succeed in under 48 h for *ao* and *wl* graphs. Orkut results are omitted due to the excessively long runtimes when using the inverse degree ordering of vertices.

input graph is preloaded into the main memory and interleaved across all NUMA regions. We remove all self-loops from the graphs and transform all directed edges to undirected edges. When evaluating the performance of the MCE implementations, we do not store the maximal cliques found, but simply count them.

3.5.2 Evaluation of Vertex Ordering Strategies

Figure 3.10 shows the impact of different vertex ordering strategies on the single-threaded execution time of the BK algorithm on Intel KNL. We evaluated three main strategies: *i)* degeneracy ordering [ELS10], *ii)* degree ordering (ascending order), and *iii)* inverse degree ordering (descending order). The inverse degree ordering strategy provides us with a lower bound of the worst-case behavior of arbitrary vertex orderings covered in Section 3.2.5. As expected, this strategy leads to the worst performance results, resulting in an up to 16× slow-down with respect to degeneracy ordering, this confirms the results of our theoretical analysis provided in Table 3.1. However, despite the better worst-case complexity bound it achieves, the degeneracy ordering does not always lead to a better practical performance than the degree ordering. We believe that further theoretical analysis could shed more light on this empirical observation.

3.5.3 Hash Joins versus Merge Joins

In this section, we evaluate the impact of set intersection algorithms on the overall performance and memory usage of the BK algorithm. The hash-join-based set intersections are implemented using *SimpleHashSet* described in Section 3.3, and the merge-join-based set intersections are implemented using CAlist described in Section 3.3.1. We also evaluate the impact of using recursive subgraph creation described in Section 3.2.2. However, instead of creating a subgraph per call, we create a subgraph per task to limit the overhead of subgraph creation.

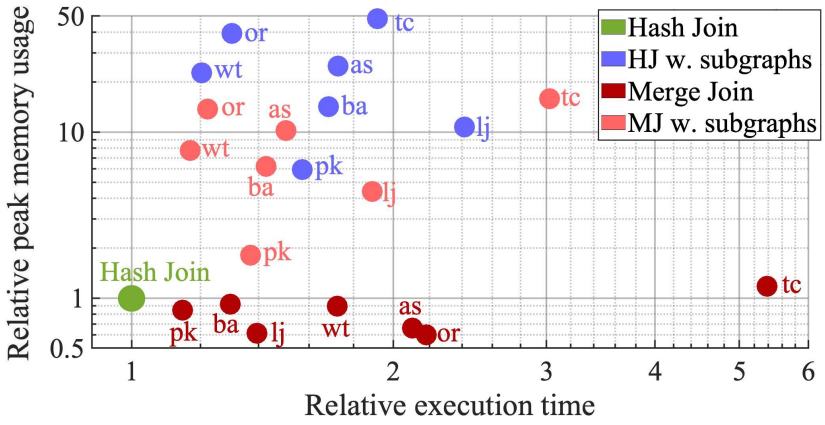


Figure 3.11: Impact of (i) hash-join- vs. merge-join-based intersection algorithms and of (ii) recursive subgraph creation on runtime and memory usage of the BK algorithm. The solution based on hash joins is Pareto-optimal.

The experiments are performed on Intel KNL using all 256 hardware threads. In Figure 3.11, we evaluate all six small graphs and one large graph from Table 3.2 and show the execution time and the peak dynamic memory usage results of different set intersection methods. We repeat these experiments both with and without recursive subgraph creation. The results given in Figure 3.11 are relative to our hash-join-based BK implementation that does not use recursive subgraph creation. Peak dynamic memory usage is measured as described in Section 3.4, where we track dynamic memory allocations and deallocations of each thread and sample the peak after a certain number of allocations. Note that these results do not include the memory used by the input graphs.

We see that the merge-join-based approach benefits from creating subgraphs. Its execution time is reduced as much as 44% for the *tc* graph, but at the cost of increased memory usage. In all the cases, our hash-join-based BK implementation that does not create subgraphs is Pareto-optimal as predicted by our theoretical analysis (see Figure 3.1). On the other hand, also as predicted by our theoretical analysis, the hash-join-based BK implementation does not benefit from recursive subgraph creation. Note that the subgraphs created by the hash-join-based implementation use more memory than those created by the merge-join-based implementation because the adjacency lists of the subgraphs are stored as hash tables in the hash-join case, and our *SimpleHashSet* implementation sets the size of the hash tables to twice the size of the adjacency lists to minimise hash conflicts.

3.5.4 Scalability Analysis

The scalability analysis is carried out on both hardware platforms shown in Table 3.3 using the graphs from Table 3.2. Figure 3.12a shows the performance improvements we achieve with respect to single-threaded execution when increasing the number of threads on the KNL architecture. We observe an almost linear performance scaling up to 64 threads, which is

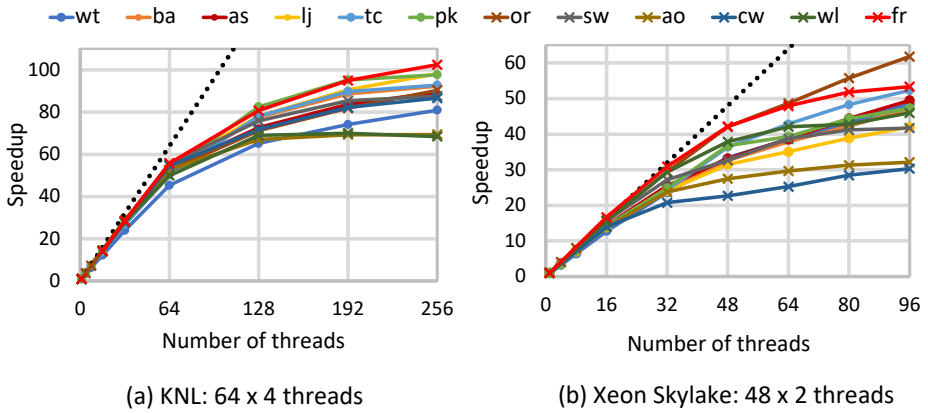


Figure 3.12: Performance scaling on modern many-core processors: speed-ups are relative to single-threaded execution. An almost linear scaling with the number of physical cores is observed.

the number of physical cores available. After 64 threads, the performance scales sublinearly because the threads running on the same core start sharing hardware resources, e.g., SIMD units. Using two hardware threads per core improves the performance by only 40%. Using all 256 hardware threads, we achieve up to 100× speedup compared to single-threaded execution. Figure 3.12b shows the performance scaling when using Xeon Skylake processors. We observe up to 60× speedup using 96 hardware threads.

3.5.5 Comparisons with the State of the Art

We compare our BK implementation with the following state-of-the-art MCE implementations: *i)* **QFilterMCE** by Han et al. [HZY18] is an optimised single-threaded implementation that uses QFilter and SIMD Galloping with BSR methods, described in Section 3.3, to accelerate set intersections. QFilterMCE uses compressed bit-vectors for representing the graph vertices and requires a preprocessing step that reorders the vertices in order to perform more efficient set intersections. *ii)* **ParMCE** by Das et al. [DST20] is an optimised shared-memory parallel C++ implementation that uses the Intel TBB library for parallelization. In the comparisons, we use our BK implementation based on degree ordering because the performance of the BK algorithm based on degree ordering is similar to the one that uses degeneracy ordering as shown in Figure. 3.10. In addition, computing the degree order does not require any advanced preprocessing.

We compare the single-threaded execution of our implementation with QFilterMCE. Figure 3.13a shows that our implementation, which uses the *SimpleHashSet* algorithm given in Section 3.3 to accelerate set intersections, displays a competitive performance to that of QFilterMCE even though our solution does not require any preprocessing. On average, taking into the account both the time to preprocess the graph and to execute QFilterMCE, our solution is faster than QFilterMCE by 4.1× on KNL and by 2.7× on Xeon Skylake. In addition, the QFilter

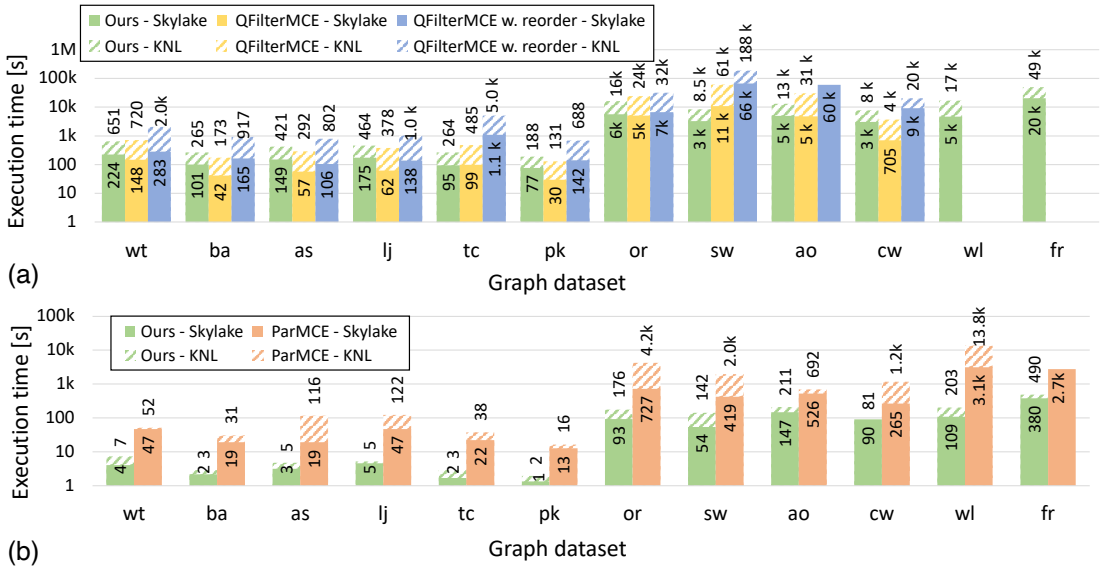


Figure 3.13: Performance of algorithms for maximal clique enumeration using (a) single thread and (b) all available hardware threads. The top value shows the execution times on the Intel KNL and the bottom value shows the execution times on the Intel Xeon Skylake. The missing values show the data points that did not execute within the given time budget.

preprocessing did not finish under 48 h on KNL for the *ao* and *wl* graphs, and under 24 h on Xeon Skylake for the *wl* graph. It also failed to execute for the *fr* graph on both platforms. Note that we reused the preprocessing results computed by Xeon Skylake for the *ao* graph when executing QFilterMCE on KNL.

Figure 3.13b compares the execution time of our manycore *SimpleHashSet*-based implementation with ParMCE using 256 hardware threads on KNL and 96 hardware threads on Xeon Skylake. On average, we achieve 14.3× and 8.3× lower execution times than ParMCE on KNL and Skylake, respectively. Note that ParMCE runs out of memory when executing the *fr* graph on KNL and goes into swap space when executing the *wl* graph on KNL. The highest speedups we achieve with respect to ParMCE on KNL and Skylake are 68× and 28×, respectively. The primary reasons for the performance improvement are SIMD-accelerated set intersections described in Section 3.3 and our manycore optimisations described in Section 3.4.

3.6 Related Work

Maximal clique enumeration (MCE) represents an important graph mining problem with applications in various fields, such as bioinformatics [YZT14; Yu+06], social network analysis [LWN18], and electronic design automation [RA12; VBI10]. The most efficient class of MCE algorithms are based on backtracking search, such as the algorithm from Bron and Kerbosch [BK73], and its improvements by Tomita et al. [TTT06] and Eppstein et al. [ELS10], which we discuss in more detail in Section 2.4.1. These algorithms do not analyze the effect of using different intersection strategies on the overall time complexity. Our work analyzes the

effect of using hash and merge joins for intersections on the time and space complexity of the algorithm by Eppstein et al. [ELS10] and reaps the corresponding advantage.

SIMD-accelerated set-intersection algorithms can be used to improve the speed of MCE [HZY18; SWL11; IOT14]. Schlegel et al. [SWL11] and Inoue et al. [IOT14] exploit STTNI instructions in order to accelerate set intersections using merge joins. QFilter by Han et al. [HZY18] further improves the performance of set intersections by using a compressed bit-vector representation. QFilter is used for accelerating graph algorithms such as MCE. However, it requires a preprocessing of the input graph in order to achieve high performance. Our *SimpleHashset* approach uses hash joins rather than merge joins, and it does not involve any significant preprocessing, yet it achieves a performance comparable to that of QFilter.

Of particular interest to our work are manycore implementations of MCE [DST20; Les+17; Sch+09]. Schmidt et al. [Sch+09] present a parallel variant of the MCE algorithm by Bron and Kerbosch [BK73], and describe a work-stealing method for load balancing. Das et al. [DST20] present ParMCE, a shared-memory parallel algorithm for MCE, which uses Intel TBB library for load balancing. ParMCE is based on the algorithm by Tomita et al. [TTT06], an improved version of the algorithm by Bron and Kerbosch [BK73]. Our work further improves the performance by addressing the task and memory management overheads that arise in our TBB-based implementation of MCE. Lessley et al. [Les+17] describe a parallel algorithm based on data-parallel primitives [Ble90] that can be executed on both manycore CPUs and GPUs by generating the corresponding TBB or CUDA code. However, it explores the search space of MCE in breadth-first order, which is memory inefficient compared to depth-first-based solutions such as ours. As pointed out by Das et al. [DST20], the CPU implementation of Lessley et al. [Les+17] fails to execute on even moderately sized graphs such as *wiki-talk* from Table 3.2. Our work takes into account the dynamic memory usage of manycore MCE and proposes methods to minimise it. Furthermore, a recent work by Almasri et al. [Alm+22; Alm22] showed that a GPU implementation of MCE that explores the search space of MCE in depth-first order instead of breadth-first order enables it to process larger graphs, such as the ones from Table 3.2.

Distributed implementations of MCE have also been proposed [SMT15; Hou+16; Che+16]. Svendsen et al. [SMT15] use different vertex ordering strategies for statically balancing the load across the computation nodes. Brighen et al. [Bri+19] propose using a vertex-centric framework Giraph [Sak+16], which is based on the bulk synchronous parallel (BSP) model [Val90], for distributed computation of MCE. However, our work uses a framework with dynamic load balancing, which is designed to cope with load balancing and synchronization issues better than the simpler static load balancing and the less specialized BSP on shared-memory manycore processors. The work by Chen et al. [Che+16] uses the idea of recursive subgraph partitioning in order to distribute the work across multiple computing nodes dynamically. This technique aims at a coarser grain parallelism than the one we exploit in our manycore implementation and is essentially orthogonal to our work.

3.7 Conclusions

In this chapter, we explore the use of join algorithms for accelerating set intersections in the MCE algorithm proposed by Eppstein et al. [ELS10]. We theoretically show that the use of hash-join-based set intersections enables Pareto-optimal MCE implementations in terms of time and space complexity compared to various possibilities that use merge-join-based set intersections. Building on this result, we introduce a simple SIMD-accelerated hash-join-based set intersection implementation and use it to accelerate MCE. Using this simple approach, we match the single-threaded performance of an MCE implementation that uses highly-optimised set intersections, which requires some time-consuming preprocessing; our implementation does not suffer from such a requirement. In addition, we contribute a many-core version of MCE that uses a shared-memory parallel processing framework for exploiting task-level parallelism. When implemented in a naive way, the many-core implementation suffers from scalability overheads and poor dynamic memory management. By addressing these issues, we achieve a maximum speedup of 100× compared to the single-threaded case on a machine with 64 physical cores; we outperform a state-of-the-art manycore MCE implementation by an order of magnitude.

4 Fine-grained Parallelisation of Cycle Enumeration Algorithms

The acceleration of simple cycle enumeration algorithms is a more challenging problem compared to the task of accelerating maximal clique enumeration. As demonstrated in the previous chapter, the prevalence of set intersection operations in the state-of-the-art sequential algorithm for maximal clique enumeration [ELS13] offers the opportunity for exploiting data parallelism in this algorithm, thus enabling its acceleration using vector instructions. In contrast, simple cycle enumeration algorithms have limited opportunities to take advantage of data parallelism because their computation primarily entails pointer chasing [Tie70; Joh77; SL76; RT75]. In addition, the method introduced in the previous chapter for the scalable parallelisation of maximal clique enumeration cannot be applied to the state-of-the-art algorithm for simple cycle enumeration by Johnson [Joh77] and its derived algorithms for enumerating simple cycles under temporal [KC18] and hop [Pen+19] constraints without incurring significant performance penalties. Furthermore, applying the straightforward coarse-grained parallelisation method, introduced in Section 1.2, to the cycle enumeration algorithms can lead to a workload imbalance across threads, as shown in Figures 1.3 and 1.4, thus limiting their scalability.

The focus of this chapter is on addressing the aforementioned problems and enabling scalable parallelisation of state-of-the-art sequential algorithms for enumerating simple, temporal, and hop-constrained cycles. First, we focus on the simple cycle enumeration problem and parallelise the algorithms by Johnson [Joh77] and by Read and Tarjan [RT75] in a fine-grained manner. These algorithms were chosen because they achieve the lowest time complexity bounds reported among the simple cycle enumeration algorithms for directed graphs [MD76; Gro16]. We theoretically show that our resulting fine-grained parallel algorithms are scalable, with the fine-grained parallel Read-Tarjan algorithm being strongly scalable. In contrast, we theoretically prove that coarse-grained parallel versions of these simple cycle enumeration algorithms are not scalable. Next, we adapt our fine-grained approach to enable the

This chapter is based on the work published at the *34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* [BIA22].

Table 4.1: Our fine-grained parallel Read-Tarjan algorithm is the only solution that is both work-efficient and scalable.

Parallel algorithm	Work-efficient	Scalable
Coarse-grained parallel algorithms	✓	
Our fine-grained parallel Johnson		✓
Our fine-grained parallel Read-Tarjan	✓	✓

enumeration of cycles under time-window, temporal, and hop constraints. Imposing such constraints further reduces the execution time of the cycle enumeration algorithms by decreasing the number of simple cycles that are enumerated. Finally, we evaluate the parallel cycle enumeration algorithms on a cluster with 256 CPU cores that can execute up to 1024 simultaneous threads, and demonstrate a near-linear scalability and an order of magnitude speedup compared with the coarse-grained parallel versions of those algorithms.

The rest of this chapter is organised as follows. Section 4.1 gives an overview of the solution introduced in this chapter. Section 4.2 presents the theoretical analysis of the coarse-grained parallel versions of the Johnson and the Read-Tarjan algorithms. Section 4.3 and Section 4.4 introduce our fine-grained parallel versions of the Johnson and the Read-Tarjan algorithms, respectively. Our general framework for parallelising temporal and hop-constrained cycle enumeration algorithms is presented in Section 4.5. In Section 4.6, we provide an experimental evaluation of our fine-grained parallel algorithms. The related work is presented in Section 4.7. Section 4.8 concludes this chapter.

4.1 Overview of the Solution

To address the issues that arise from coarse-grained parallelisation of the simple cycle enumeration algorithms, we propose a fine-grained parallel version of the Johnson algorithm. The proposed fine-grained parallelisation enables the scalable execution of the Johnson algorithms by decomposing the long sequential searches into fine-grained tasks, which are then dynamically scheduled across CPU cores. As explained in Section 1.2, the pruning efficiency of the Johnson algorithm depends on the strict depth-first-search-based traversal of its recursion trees, making it challenging to parallelise in a fine-grained manner. To decompose the Johnson algorithm into fine-grained tasks, we propose the *copy-on-steal* mechanism that relaxes the strictly depth-first-search-based exploration the Johnson algorithm performs, enabling this algorithm to perform multiple independent depth-first searches in parallel. As a result, our fine-grained parallel Johnson algorithm is able to achieve ideal load balancing and near-linear performance scaling.

As an alternative approach for achieving fast and scalable simple cycle enumeration, we focus on the fine-grained parallelisation of the lesser-known Read-Tarjan algorithm for simple cycle enumeration. This algorithm has the same theoretical worst-case complexity as the Johnson algorithm (see Section 2.4.2), which has the lowest complexity among the related algorithms

Table 4.2: Capabilities of the related work versus our own. Competing algorithms either fail to exploit fine-grained parallelism or do it on top of asymptotically inferior algorithms.

Related work	**[KC18]**	**[Qiu+18]**	**[Pen+19]**	**[Qin+20]**	**[GS21]**	**Ours**
Fine-grained parallelism				✓		✓
Asymptotic optimality	✓		✓		✓	✓
Temporal ordering constraints	✓					✓
Time-window constraints	✓	✓				✓
Hop constraints		✓	✓	✓	✓	✓

for simple cycle enumeration [MD76; Gro16]. We demonstrate that the Read-Tarjan algorithm is easier to decompose into fine-grained tasks compared to the Johnson algorithm because the Read-Tarjan algorithm does not require the strict depth-first-search-based exploration of its recursion trees. Similarly to our fine-grained parallel Johnson algorithm, the resulting fine-grained parallel Read-Tarjan algorithm is able to achieve an almost linear scaling of performance with the number of CPU cores utilised. However, the Johnson algorithm is faster than the Read-Tarjan algorithm in practice despite having the same theoretical time complexity [Gro16; MD76], which can be observed when comparing the performance of our fine-grained parallel versions of these algorithms. The reason for this behaviour is more aggressive pruning technique employed by the Johnson algorithm, as demonstrated in Figure 2.4. To make the Read-Tarjan algorithm competitive with the Johnson algorithm, we have introduced several optimisations that enhance the pruning efficiency of the Read-Tarjan algorithm. These optimisations reduce the number of unnecessary vertex visits that this algorithm performs and enable up to 6.8× faster execution of the Read-Tarjan algorithm.

The theoretical analysis presented in this chapter shows that both of our fine-grained parallel algorithms are scalable, which is not the case for the Johnson and the Read-Tarjan algorithms parallelised in a coarse-grained manner. Moreover, it shows that our fine-grained parallel Read-Tarjan algorithm performs asymptotically the same amount of work as its serial version, whereas our fine-grained parallel Johnson algorithm does not. Therefore, our fine-grained parallel Read-Tarjan algorithm is the only parallel algorithm based on an asymptotically-optimal cycle enumeration algorithm that is both work-efficient and scalable, as shown in Table 4.1. Interestingly, despite not being work-efficient, our fine-grained Johnson algorithm outperforms our fine-grained parallel Read-Tarjan algorithm in most of our experiments.

To reduce the computational complexity of the cycle enumeration algorithms, different types of constraints are often imposed during the search for simple cycles, as shown in Table 4.2. Examples of these constraints are *temporal ordering* constraints, which reduce the search to temporal cycles only [KC18], *hop* constraints [Pen+19; Qiu+18], which limit the length of paths explored during the search for cycles, and *time-window* constraints [KC18], which restrict the search to cycles that occur within a time window of a given size. Imposing these constraints reduces the number of paths explored during the search for cycles, making the problem more tractable. For this reason, all algorithms presented in this chapter are adapted to enable the enumeration of cycles under time-window constraints. In addition, we show that our method

Table 4.3: Work and depth of the coarse- and fine-grained parallel algorithms.

Parallel algorithm	Work	Depth
Coarse-grained algorithms	$O(n+e+ec)$	$O(ec)$
Fine-grained Johnson algorithm	$O\big(n+e+\min\{pce, se\}\big)$	$O(e)$
Fine-grained Read-Tarjan algorithm	$O(n+e+ec)$	$O(ne)$

for parallelising the Johnson algorithm in a fine-grained manner can be adapted to parallelise the state-of-the-art algorithms for temporal and hop-constrained cycle enumeration. This adaptation is possible because these state-of-the-art algorithms, such as the 2SCENT algorithm for temporal cycle enumeration [KC18] and the BC-DFS algorithm [Pen+19] for hop-constrained cycle enumeration, are extensions of the Johnson algorithm. Regardless of the type of constraint used, our fine-grained versions of the Johnson and the Read-Tarjan algorithms are an order of magnitude faster than the straightforward coarse-grained parallel versions of these algorithms when executed on a system that can execute up to a thousand concurrent software threads.

4.2 Coarse-Grained Parallel Methods

The most straightforward way of parallelising the Johnson and the Read-Tarjan algorithms is to search for cycles that start from different vertices in parallel. Each such search can then be executed by a different thread that explores its own recursion tree. This approach is beneficial because it is work-efficient and can be implemented using one of the existing graph processing frameworks, such as Pregel [Mal+10], in a manner similar to the method used by Rocha and Thatte [RT15]. We refer to this parallelisation approach as the coarse-grained parallel approach.

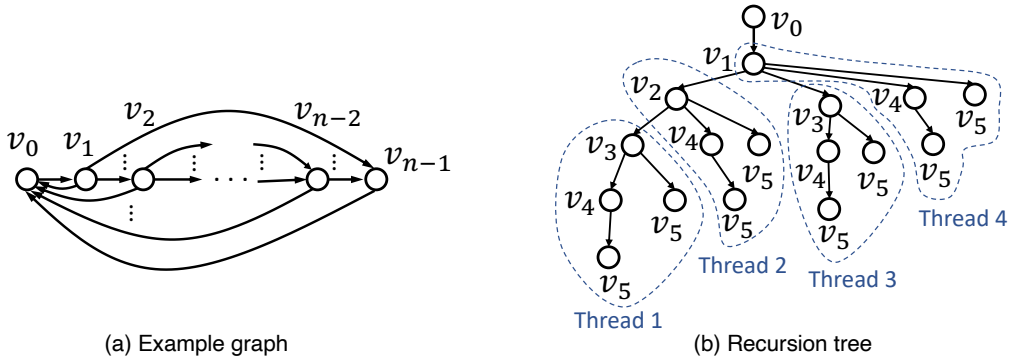
The coarse-grained approach can express more parallelism if each thread performs a search for cycles that start from a different edge rather than a different vertex. This assumption is supported by the fact that graphs typically have more edges than vertices. Nevertheless, the coarse-grained approach for parallelising the simple cycle enumeration algorithms is not scalable, which we prove here.

Proposition 1. *The coarse-grained parallel Johnson and Read-Tarjan algorithms are work-efficient.*

The proof of Proposition 1 is trivial, and we omit it for brevity.

Theorem 7. *The coarse-grained parallel Johnson and Read-Tarjan algorithms are not scalable.*

Proof. In this case, the depth $T_\infty(n)$ represents the worst-case execution time of a search for cycles that starts from a single vertex or edge, and it depends on the number of cycles found during this search. In the worst case, a single recursive search can discover all cycles of a graph. An example of such graph is given in Figure 4.1a, where each vertex v_i, with



(a) Example graph (b) Recursion tree

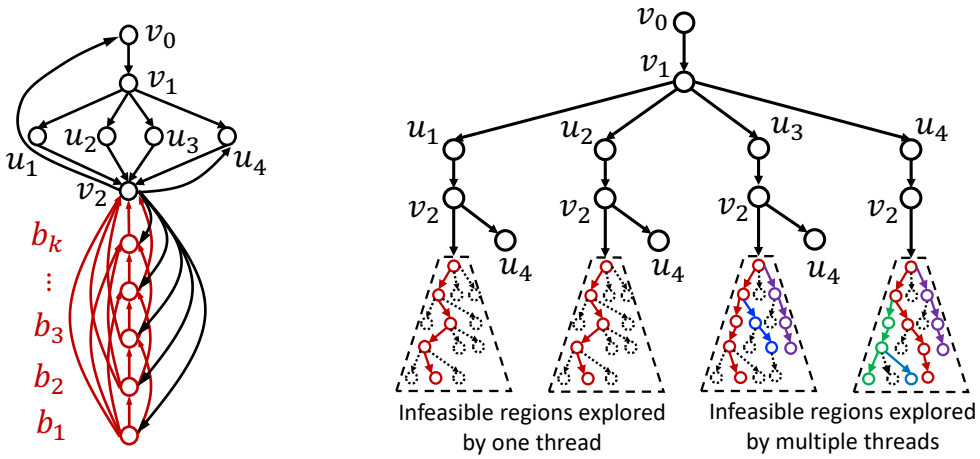
Figure 4.1: (a) A graph with an exponential number of simple cycles. (b) The recursion tree of the Johnson algorithm for $n = 6$ constructed when the algorithm starts from v_0. Whereas a coarse-grained parallel algorithm explores the complete recursion tree using a single thread, our fine-grained parallel algorithms can explore different regions of the recursion tree in parallel using several threads.

$i \in \{1, \ldots, n-1\}$, is connected to v_0 and to every vertex v_j such that $j > i$. In that graph, any subset of vertices $v_2, \ldots, v_{n-1}$ defines a different cycle. Therefore, the total number of cycles in this graph is equal to the number of all such subsets $c = 2^{n-2}$. Before the search for cycles, both the Johnson and the Read-Tarjan algorithm find all vertices that start a cycle, which is only v_0 in this case. Therefore, the search for cycles will be performed only by one thread. Because both the Johnson and the Read-Tarjan algorithms require $O(e)$ time to find each cycle, the depth of the coarse-grained algorithms is $T_\infty(n) \in O(ec)$. Because $\lim_{n\to\infty} T_\infty(n)/T_1(n) \neq 0$, the coarse-grained algorithms are not scalable based on Definition 4. □

Theorem 7 shows that the main drawback of the coarse-grained parallel algorithms is their limited scalability. This limitation is apparent for the graph shown in Figure 4.1a, which has an exponential number of cycles in n. When using a coarse-grained parallel algorithm on this graph, all the cycles will be discovered by a single thread, and, thus, the depth of this algorithm grows linearly with c, as shown in Table 4.3. Because only one thread can be effectively utilised, increasing the number of threads will not result in a reduction of the overall execution time of the coarse-grained parallel algorithm. Figure 1.4 shows the workload imbalance exhibited by the coarse-grained parallel algorithms in practice. Section 5.5 demonstrates the limited scalability of coarse-grained parallel algorithms in further detail.

4.3 Fine-Grained Parallel Johnson Algorithm

To address the load imbalance issues that manifest themselves in the coarse-grained parallel Johnson algorithm, we introduce the *fine-grained parallel Johnson* algorithm. The main goal of our fine-grained algorithm is to enable several threads to explore a recursion tree concurrently, as shown in Figure 4.1b, where each thread executes a subset of the recursive calls of this tree. However, enabling concurrent exploration of a recursion tree is in conflict with the sequential



(a) Example graph (b) Recursion tree

Figure 4.2: (a) An example graph and (b) the recursion tree of our fine-grained Johnson algorithm when enumerating cycles that start from v_0. Each thread of our fine-grained Johnson algorithm explores the vertices $b_1, \ldots, b_k$ at most once.

depth-first exploration, required by the Johnson algorithm to achieve a high pruning efficiency.

In this section, we first discuss the challenges that arise when parallelising the exploration of a recursion tree of the Johnson algorithm. Then, we introduce the *copy-on-steal* mechanism used to address these challenges and present our fine-grained parallel Johnson algorithm. Finally, we theoretically analyse our algorithm and show that it is scalable.

4.3.1 Fine-Grained Parallelisation Challenges

The requirement of the sequential depth-first exploration of the Johnson algorithm makes it challenging to efficiently parallelise this algorithm in a fine-grained manner. This requirement is enforced by maintaining a set of blocked vertices *Blk* throughout the exploration of a recursion tree. If threads exploring the same recursion tree simply share the same set of blocked vertices *Blk*, the parallel algorithm could produce incorrect results. For example, considering the graph given in Figure 4.2a, a thread exploring the path $\Pi = v_0 \rightarrow v_1 \rightarrow u_1 \rightarrow v_2$ visit and block the vertex u_4 in this case because u_4 cannot participate in a simple cycle that begins with Π. Because the threads exploring this graph share the blocked vertices, another thread attempting to discover the cycle $v_0 \rightarrow v_1 \rightarrow u_4 \rightarrow v_2 \rightarrow v_0$ would fail to do so because u_4 is blocked. Therefore, this approach might not discover all cycles in a graph.

To enable several threads to correctly find all cycles while exploring the same recursion tree, the algorithm could forward a new copy of the *Blk* and *Blist* data structures when invoking each child recursive call. However, this approach would redundantly explore many paths in a graph. The reason is that a recursive call would be unaware of the vertices visited and blocked by other calls that precede it in the depth-first order except for its direct ancestors in the recursion tree. When enumerating the simple cycles of the graph shown in Figure 4.2a

Algorithm 8: FGJ_task (v, v_0, $\mathcal{G}$, d, T_1)

Input: v - the current vertex, v_0 - the starting vertex
$\mathcal{G}$ - the input graph
d - the depth of this task
InOut: T_1 - the thread that created this task ▷ Maintains Π_{T_1}, Blk_{T_1}, $Blist_{T_1}$, and $Mutex_{T_1}$
Output: *true* if a cycle was found

```
1  T2 = the thread executing this task;          ▷ Maintains Π_T2, Blk_T2, Blist_T2, and Mutex_T2
2  if T1 ≠ T2 then FGJ_copyOnSteal(d, T1, T2);          ▷ Check if this task is stolen
3  Mutex_T2.lock();
4  Π_T2.push(v); Blk_T2 = Blk_T2 ∪ {v};
5  Mutex_T2.unlock();
6  found = false;
7  foreach u : N_G(v) s.t. u.id > v0.id do          ▷ Recursively explore the neighbours of v
8      if u = v0 then
9          report cycle Π_T2;
10         found = true;
11     else if u ∉ Blk_T2 then
12         f = spawn FGJ_task(u, v0, G, d + 1, T2) ;          ▷ Create a child task
13         found = found ∨ f;
14 sync;                                          ▷ Wait for the spawned tasks
15 Mutex_T2.lock();
16 Π_T2.pop();
17 if found then                                  ▷ Unblock vertices if a cycle was found
18     RecursiveUnblock(v, Blk_T2, Blist_T2);
19 else
20     foreach u : N_G(v) do Blist_T2[u] = Blist_T2[u] ∪ {v};
21 Mutex_T2.unlock();
22 return found;
```

starting from v_0, this approach explores all $4 \times 2^{k-1} + 3$ maximal simple paths instead of just seven, that the Johnson algorithm would explore. Hence, this approach exhaustively explores all maximal simple paths in the graph and is identical to the brute-force solution of Tiernan (see Section 2.4.2). Next, we propose a fine-grained parallel algorithm that addresses the aforementioned parallelisation challenges.

4.3.2 Copy-on-Steal

To enable different threads to concurrently explore the recursion tree in a depth-first fashion while also taking advantage of the powerful pruning capabilities of the Johnson algorithm, each thread executing our fine-grained parallel Johnson algorithm maintains its own copy of the Π, *Blk*, and *Blist* data structures. These data structures are copied between threads only when these threads attempt to explore the same recursion tree. To achieve this behaviour,

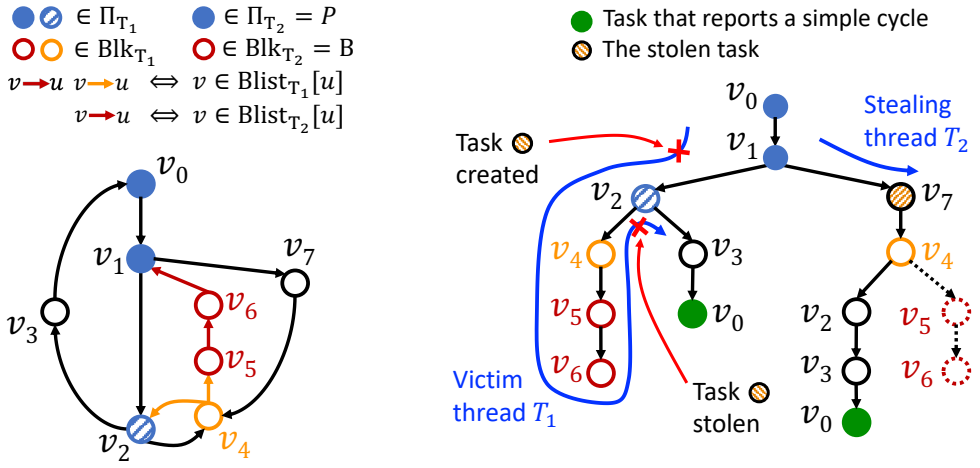


Figure 4.3: (a) An example graph and (b) the recursion tree of our fine-grained Johnson algorithm when enumerating simple cycles that start from v_0. Here, X_{T_i} denotes a data structure X of the thread T_i. The thread T_2 can prune the dotted part of the tree by avoiding v_5 and v_6 that the thread T_1 has blocked after creating the task stolen by T_2.

our fine-grained parallel Johnson algorithm implements each recursive call of the Johnson algorithm as a separate task. The pseudocode of this task is given in Algorithm 8, where a data structure X, maintained by the thread T_i, is denoted as X_{T_i} (see Table 2.1). If a child task and its parent task are executed by the same thread T_i, the child task reuses the Π_{T_i}, Blk_{T_i}, and $Blist_{T_i}$ data structures of the parent task. However, if a child task has been stolen—i.e., it is executed by a thread other than the thread that created it, the child task will allocate a new copy of these data structures (line 2 of Algorithm 8). We refer to this mechanism as *copy-on-steal*.

The problem with copying data structures between different threads upon task stealing is that the thread that has created the stolen task (i.e., the *victim thread*) can modify its data structures before this task is stolen by another thread (i.e., the *stealing thread*). This problem can be observed in the example shown in Figure 4.3. There, the victim thread T_1 and the stealing thread T_2 explore the same recursion tree given in Figure 4.3b while searching for cycles that start with $P_1 = v_0 \rightarrow v_1 \rightarrow v_2$ and $P_2 = v_0 \rightarrow v_1 \rightarrow v_7$, respectively. In this case, T_2 steals a task created by T_1 that explores v_7, as indicated in Figure 4.3b, and receives a copy of the blocked vertices $Blk_{T_1} = \{v_4, v_5, v_6\}$ discovered by T_1. The thread T_1 blocked these vertices because they cannot participate in any simple cycle that begins with P_1. If T_2 simply uses a copy of these blocked vertices Blk_{T_1} without modifications, T_2 will be unable to find the cycle $v_0 \rightarrow v_1 \rightarrow v_7 \rightarrow v_4 \rightarrow v_2 \rightarrow v_3 \rightarrow v_0$ because v_4 is blocked. Therefore, a method for unblocking vertices after copy-on-steal is required to correctly find all cycles.

We explore two solutions for this problem:

(i) Copy-on-steal with complete unblocking. To enable the threads of our algorithm to find cycles after performing copy-on-steal, the stealing thread could unblock all vertices that the

Algorithm 9: FGJ_copyOnSteal (d, T_1, T_2)

Input: d - the depth of the task executing this function
InOut: T_1 - the victim thread
T_2 - the stealing thread

```
1 Mutex_T1.lock();
2 {Π_T2, Blk_T2, Blist_T2} = copy({Π_T1, Blk_T1, Blist_T1});   ▷ Copy the data of T1 to T2
3 Mutex_T1.unlock();
4 while |Π_T2| ≥ d do                                          ▷ Copy-on-steal with recursive unblocking
5 |   u = Π_T2.pop();
6 |   RecursiveUnblock(u, Blk_T2, Blist_T2);
```

victim thread had blocked after creating the stolen task. In our example given in Figure 4.3, the stealing thread T_2 unblocks all vertices $Blk_{T_1} = \{v_4, v_5, v_6\}$ it received from the victim thread T_1. Although this approach enables T_2 to correctly find cycles, it also fails to take advantage of the information collected by T_1 to reduce the redundant work of T_2. For instance, in Figure 4.3, T_2 visits v_5 and v_6, even though T_1 already concluded that these vertices cannot participate in any simple cycle that begins with $P = v_0 \rightarrow v_1$, where P is the largest common prefix of all the paths explored by T_1 and T_2. As a result, T_2 redundantly visits the dotted part of the recursion tree given in Figure 4.3b.

(ii) Copy-on-steal with recursive unblocking. In this approach, the stealing thread capitalises on the information already discovered by the victim thread. The stealing thread T_2 can reuse a subset $B \subset Blk_{T_1}$ of the blocked vertices discovered by T_1 if the vertices in B cannot participate in simple cycles that begin with P, where P is the largest common prefix of all the paths explored by T_1 and T_2. Because any path discovered by T_2 begins with P, T_2 can avoid visiting vertices from B. Thus, to correctly find simple cycles, it is sufficient for T_2 to unblock the vertices from $Blk_{T_1} \setminus B$. To achieve this behaviour, T_2 invokes a recursive unblocking procedure of the Johnson algorithm (see lines 17–21 Algorithm 4) for every vertex $v \in \Pi_{T_1} \setminus P$, as shown in Algorithm 9, where Π_{T_1} is the path T_1 is exploring during task stealing. The vertices in B can only be unblocked by a recursive unblocking invoked for $v \in P$; hence, the vertices in B remain blocked. In the example given in Figure 4.3, T_2 invokes a recursive unblocking procedure for $\Pi_{T_1} \setminus P = \{v_2\}$, which results in unblocking of v_4. Thus, T_2 is able to discover a cycle that contains v_4. The vertices $B = \{v_5, v_6\}$ will not be unblocked because they cannot take part in any simple cycle that begins with $P = v_0 \rightarrow v_1$. Therefore, thread T_2 avoids visiting the dotted part of the recursion tree given in Figure 4.3b.

Without countermeasures, our algorithm can suffer from race conditions because its data structures can be accessed concurrently by different threads. For instance, a stealing thread T_2 can copy the data structures of a victim thread T_1 while T_1 performs a recursive unblocking, in which case T_2 could receive the vertex set Blk_{T_1} that is partially unblocked. When using copy-on-steal with recursive unblocking, T_2 may not be able to continue the interrupted unblocking of Blk_{T_1}, causing the algorithm to miss certain cycles. To avoid this problem, we define critical sections in lines 15–21 of Algorithm 8 and in lines 1–3 of Algorithm 9 using coarse-grained

Algorithm 10: FGJ ($\mathcal{G}(\mathcal{V},\mathcal{E})$)

Input: $\mathcal{G}$ - the input graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$

```
1 parallel foreach v_0 → v : E do
2     T_0 = the thread executing this iteration;   ▷ Maintains Π_T0, Blk_T0, Blist_T0, and Mutex_T0
3     Π_T0 = v_0;  Blk_T0 = ∅;
4     foreach u : V do Blist_T0[u] = ∅;
5     spawn FGJ_task(v, v_0, G, 1, T_0);                                  ▷ Create a task
6 sync;                                                                   ▷ Wait for all spawned tasks
```

locking by maintaining a mutex per thread. However, such a locking mechanism is not required when using copy-on-steal with complete unblocking because T_2 can correctly unblock vertices in Blk_{T_1} simply by removing all vertices from Blk_{T_1} inserted after the stolen task was created. Thus, it is sufficient to enable thread-safe operations on Π, *Blk*, and *Blist* using fine-grained locking. As a result, the critical sections are shorter when the copy-on-steal with complete unblocking approach is used.

Nevertheless, we opt to use the copy-on-steal with recursive unblocking approach in our fine-grained parallel Johnson algorithm because this approach leads to less redundant work and rarely suffers from synchronisation overheads. The pseudocode of our fine-grained parallel Johnson algorithm is given in Algorithm 10.

4.3.3 Theoretical Analysis

We now show that the fine-grained parallel Johnson algorithm is scalable but not work-efficient.

Theorem 8. *The fine-grained parallel Johnson algorithm is not work-efficient.*

Proof. According to Lemma 3 presented by Johnson [Joh75], a vertex cannot be unblocked more than once unless a cycle is found, and once a vertex is visited, it can be visited again only after being unblocked. Thus, the Johnson algorithm visits each vertex and edge at most c times. In the fine-grained parallel Johnson algorithm executed using p threads, each thread maintains a separate set of data structures used for managing blocked vertices. Because the threads are unaware of each other's blocked vertices, each edge is visited at most pc times, c times by each thread. Additionally, an edge cannot be visited more than s times because each maximal simple path of a graph is explored by a different thread in the worst case, and during each simple path exploration, an edge is visited at most once. Therefore, the maximum number of times an edge can be visited by the fine-grained parallel Johnson algorithm is $\min\{pc, s\}$. Because the algorithm executes in $O(n+e)$ time if there does not exist a cycle or a path in the input graph, the work performed by the fine-grained parallel Johnson algorithm is

$$W_p(n) \in O\big(n+e+\min\{pce, se\}\big). \tag{4.1}$$

When $c > 0$, $p > 1$, and $s > c$, the work performed by the fine-grained parallel Johnson algorithm $W_p(n)$ is greater than the execution time $T_1(n)$ of the sequential Johnson algorithm. Thus, this algorithm is not work-efficient. □

The work inefficiency of our fine-grained parallel Johnson algorithm occurs if more than one thread performs the work the sequential Johnson algorithm would perform between the discovery of two cycles. This behaviour can be illustrated using the graph from Figure 4.2a, which contains $c = 4$ cycles and $s = c \times 2^{k-1} + 3$ maximal simple paths, each starting from vertex v_0. When discovering each cycle, our fine-grained algorithm explores an infeasible region of the recursion tree, as shown in Figure 4.2b, in which the vertices $b_1, \ldots, b_k$ are visited. If this infeasible region is explored using a single thread, each vertex b_i, with $i \in \{1, \ldots, m\}$, will be visited exactly once. However, if p threads are exploring the same infeasible region of the recursion tree, vertices $b_1, \ldots, b_k$ will be visited up to p times because the threads are unaware of each other's blocked vertices. In this case, the fine-grained parallel Johnson algorithm performs more work than necessary, and, thus, it is not work-efficient. Additionally, each infeasible region of the recursion tree that visits vertices $b_1, \ldots, b_k$ can be executed by at most $s/c = 2^{k-1}$ threads because there are 2^{k-1} maximal simple paths that can be explored in each infeasible region. In this case, each vertex b_i, with $i \in \{1, \ldots, k\}$, is visited up to s times, and, thus, the fine-grained parallel Johnson algorithm behaves as the Tiernan algorithm (see Section 2.4.2).

Lemma 4. *The depth $T_\infty(n)$ of the fine-grained parallel Johnson algorithm is in $O(e)$.*

Proof. The worst-case depth of this algorithm occurs when a thread performs copy-on-steal and explores a maximal simple path. A thread explores such a path in $O(e)$ time because it visits at most e edges. As a result, Π and *Blk* contain at most n vertices, and *Blist* contains at most e pairs of vertices. Therefore, copy-on-steal requires $O(e)$ time to copy Π, *Blk*, and *Blist*, and to unblock vertices in *Blk*. As a result, the depth of this algorithm is $T_\infty(n) \in O(e)$. □

Theorem 9. *The fine-grained parallel Johnson algorithm is scalable when* $\lim_{n\to\infty} c = \infty$.

Proof. For this algorithm, $T_1(n) \in O(n + e + ec)$ and $T_\infty(n) \in O(e)$ (see Lemma 4). Given $e < n^2$ and our assumption that $\lim_{n\to\infty} c = \infty$, we have $\lim_{n\to\infty} \frac{T_\infty(n)}{T_1(n)} = \lim_{n\to\infty} \frac{e}{n + e + ec} = 0$. Thus, this algorithm is scalable based on Definition 4. □

For the fine-grained parallel Johnson algorithm to be scalable, it is sufficient for c to increase sublinearly with n. Even though this algorithm is scalable, a strong or weak scalability is not guaranteed due to the work inefficiency of this algorithm. Nevertheless, our experiments show that this algorithm is strongly scalable in practice (see Figure 4.13).

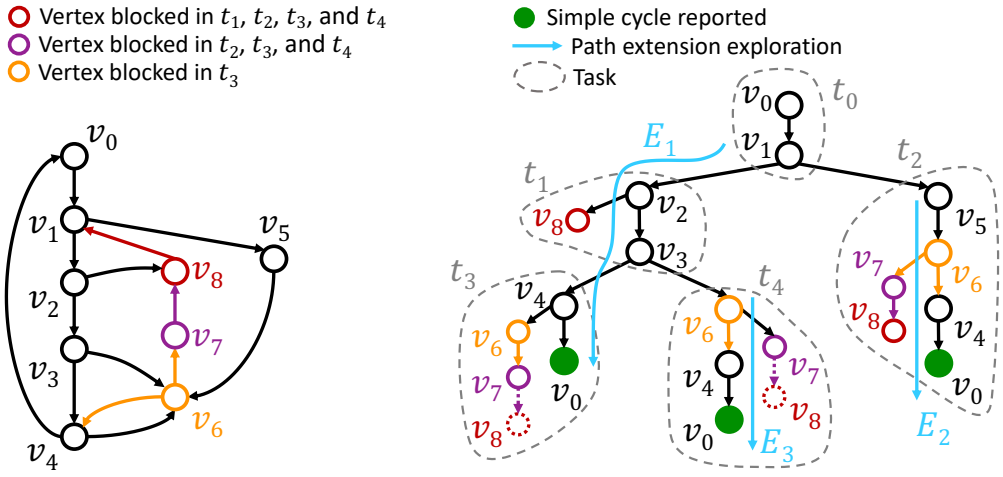


Figure 4.4: (a) An example graph and (b) the recursion tree of our fine-grained parallel Read-Tarjan algorithm when enumerating cycles that start from v_0. The nodes of the recursion tree represent the recursive calls of the depth-first search. Tasks shown in (b) can be executed independently of each other.

4.3.4 Summary

Our relaxation of the strictly depth-first-search-based recursion-tree exploration reduces the pruning efficiency of the Johnson algorithm. In the worst case, the fine-grained parallel Johnson algorithm could perform as much work as the brute-force Tiernan algorithm does—i.e., $O(se)$. However, in practice, this worst-case scenario does not happen (see Section 5.5). In addition, our fine-grained parallel Johnson algorithm can suffer from synchronisation issues in some rare cases (see Section 5.5) because our copy-on-steal mechanism can lead to long critical sections. In the next section, we introduce a fine-grained parallel algorithm that is scalable, work-efficient, and less prone to synchronisation issues.

4.4 Fine-Grained Parallel Read-Tarjan Algorithm

In this section, we first introduce several optimisations that reduce the number of unnecessary vertex visits performed by the sequential Read-Tarjan algorithm. Then, we present our fine-grained parallel Read-Tarjan algorithm that includes these optimisations. Finally, we show that our parallel algorithm is work-efficient and strongly-scalable.

4.4.1 Improvements to the Pruning Efficiency

To improve the pruning efficiency of the sequential Read-Tarjan algorithm, we include the following optimisations:

(i) Blocked vertex set forwarding enables a recursive call of the Read-Tarjan algorithm to

Algorithm 11: FGRT_DFS(u, v_0, $\mathcal{G}$, Blk, Vis)

Input: u - the current vertex, v_0 - the starting vertex
$\mathcal{G}$ - the input graph
InOut: Blk - blocked vertices
Vis - vertices visited during the DFS
Output: E - the resulting path extension from u to v_0

```
1  if u = v0 then return u;
2  Vis = Vis ∪ {u};
3  block = true;
4  foreach w : N_G(u) s.t. w.id > v0.id do
5      if w = v0 then
6          return u → w
7      else if w ∉ Blk ∧ w ∉ Vis then
8          E = FGRT_DFS(w, v0, G, Blk, Vis);   ▷ Recursively search for the path extension E
9          if E ≠ ∅ then
10             return E.push_front(u);
11     if w ∉ Blk then
12         block = false
13 if block then Blk = Blk ∪ {u};                ▷ Blocking on a successful DFS
14 return ∅;
```

reuse vertices blocked by its parent call, resulting in fewer vertex visits. The original Read-Tarjan algorithm discards blocked vertices after each recursive call [RT75], even though this information could be reused later. In this optimisation, the algorithm forwards the blocked vertices *Blk* of a recursive call to its child recursive calls, preventing those child calls from unnecessarily visiting the vertices in *Blk* again. For example, in Figure 4.4, the vertex v_8 is blocked the first time the algorithm visits v_8 while exploring the path extension E_1. This optimisation prevents the algorithm from visiting v_8 again when exploring the same extension E_1 or another extension E_3 that branches from E_1. As a result of this optimisation, the algorithm can avoid the dotted part of the recursion tree.

(ii) Path extension forwarding prevents recomputation of the path extension E found by a parent recursive call by forwarding this path extension to its child recursive call. In this way, each child recursive call performs one fewer DFS invocation than the original Read-Tarjan algorithm [RT75].

(iii) Blocking on a successful DFS is another mechanism for discovering vertices to be blocked. As a reminder, the Read-Tarjan algorithm searches for path extensions using a DFS. In the original algorithm, a vertex is blocked only if it is visited during an unsuccessful DFS invocation, which fails to discover a path extension. However, successful DFS invocations could also visit some vertices that have all their neighbours blocked. Such vertices cannot lead to the discovery of new cycles and, thus, can also be blocked. The pseudocode of the DFS function that includes this optimisation is given in Algorithm 11. In our example given in Figure 4.4, a successful

DFS invoked from v_3 finds a path extension E_3 and discovers that the only neighbour v_8 of v_7 is blocked. The algorithm then blocks v_7, which enables it to avoid visiting v_7 again when exploring E_3. Therefore, fewer vertices are visited during the execution of the algorithm.

4.4.2 Fine-Grained Parallelisation

Although the optimisations presented in Section 4.4.1 eliminate some of the redundant work performed by the Read-Tarjan algorithm, this algorithm typically performs more work than the Johnson algorithm (see Section 2.4.2). However, this redundancy makes it possible to parallelise the Read-Tarjan algorithm in a scalable and work-efficient manner.

Because the Read-Tarjan algorithm allocates a new *Blk* set for each path extension exploration, a recursive call can explore different path extensions in an arbitrary order. In addition, discovery of a new path extension E results in the invocation of a single recursive call, and these calls can be executed in an arbitrary order. As a result, several threads can concurrently explore different paths of the same recursion tree constructed by the Read-Tarjan algorithm for a given starting edge. There are neither data dependencies nor ordering requirements between different calls, apart from those that exist between a parent and a child. To exploit the parallelism available during the recursion tree exploration, we execute each path extension exploration in each recursive call as a separate task, all of which can be independently executed. Examples of such tasks are shown in Figure 4.4. We refer to the resulting algorithm as the *fine-grained parallel Read-Tarjan* algorithm.

Our implementation shown in Algorithm 12 performs only a single path extension exploration in a recursive call and uses all the optimisations we introduced in Section 4.4.1. We execute each such recursive call as a separate task using a dynamic thread scheduling framework (see Section 2.2). To find all cycles of a graph, we execute a *parallel for* loop iteration for each edge $v_0 \rightarrow v$ that uses Algorithm 11 to search for a path extension E from v to v_0, as shown in Algorithm 13 If such E exists, a task is created using v, v_0, and E as its input parameters. This task then recursively creates new tasks, as shown in lines 14 and 19 of Algorithm 12, until all cycles that start with the edge $v_0 \rightarrow v$ have been discovered.

To prevent different threads from concurrently modifying Π and *Blk*, each task allocates and maintains its own Π and *Blk* sets. A task can receive a copy of Π and *Blk* directly from its parent task at the time of task creation. However, it is possible to minimise the copy overheads by copying these sets only when a task is stolen. For this purpose, we use the copy-on-steal with complete unblocking approach described in Section 4.3.2, which has shorter critical sections than the copy-on-steal with recursive unblocking approach used by our fine-grained parallel Johnson algorithm.

Algorithm 12: FGRT_task(v, v_0, $\mathcal{G}$, E, d, T_1)

Input: v - the current vertex, v_0 - the starting vertex
$\mathcal{G}$ - the input graph
E - the path extension from v to v_0
d - the depth of this task
InOut: T_1 - the thread that created this task ▷ Maintains Π_{T_1} and Blk_{T_1}

```
1  T2 = the thread executing this task;                    ▷ Maintains Π_T2 and Blk_T2
2  if T1 ≠ T2 then                                         ▷ Check if this task is stolen
3  |  {Π_T2, Blk_T2} = copy({Π_T1, Blk_T1});              ▷ Operations on Π and Blk are thread-safe
4  while Π_T2.back() ≠ v do Π_T2.pop();
5  Remove vertices from Blk_T2 inserted at depth d' ≥ d;
6  found = false;
7  while E ≠ ∅ do                                          ▷ Exploration of the path extension E
8  |  v = E.pop_front();
9  |  Π_T2 = Π_T2.push(v); Blk_T2 = Blk_T2 ∪ {v};
10 |  foreach u : N_G(v) s.t. u.id > v0.id do
11 |  |  if u ≠ E.front() ∧ u ∉ Blk_T2 then
12 |  |  |  E' = FGRT_DFS(u, v0, Blk_T2, Vis = ∅);      ▷ Find an alternate path extension E'
13 |  |  |  if E' ≠ ∅ then
14 |  |  |  |  spawn FGRT_task(v, v0, G, E', d + 1, T2);   ▷ Create a child task
15 |  |  |  |  found = true;
16 |  |  |  else Blk_T2 = Blk_T2 ∪ Vis;
17 |  if found then break;
18 if E = ∅ then report cycle Π_T2;
19 else spawn FGRT_task(v, v0, G, E, d + 1, T2);         ▷ Create a child task
20 sync;                                                  ▷ Wait for the spawned tasks
```

4.4.3 Theoretical Analysis

We now show that the fine-grained parallel Read-Tarjan algorithm is both work-efficient and strongly scalable.

Theorem 10. *The fine-grained parallel Read-Tarjan algorithm is work-efficient.*

Proof. Because each task of our fine-grained parallel Read-Tarjan algorithm either discovers a cycle or creates at least two child tasks, our algorithm is executed using $O(c)$ tasks. Each task performs several unsuccessful DFS invocations and one successful DFS per each child task it creates. All unsuccessful DFS invocations explore at most e edges in total because they share the same set of blocked vertices. In the worst case, each edge is visited twice per task, once by a successful DFS and once by one of the unsuccessful DFS invocations. Thus, this algorithm performs $O(e)$ work per task. Because this algorithm performs $O(n+e)$ work if there are no cycles in the graph, the total amount of work this algorithm performs is $W_p(n) = O(n+e+ec)$. Hence, this algorithm is work-efficient based on Definition 3. □

Algorithm 13: FGRT ($\mathcal{G}(\mathcal{V},\mathcal{E})$)

Input: $\mathcal{G}$ - the input graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$

```
1 parallel foreach v0 → v : E do
2     T0 = the thread executing this iteration;          ▷ Maintains Π_T0 and Blk_T0
3     Π_T0 = v0; Blk_T0 = ∅;                  ▷ Operations on Π and Blk are thread-safe
4     E = FGRT_DFS(v, v0, G, Blk_T0, Vis = ∅);
5     if E ≠ ∅ then spawn FGRT_task(v, v0, G, E, 1, T0);          ▷ Create a task
6 sync;                                                  ▷ Wait for all spawned tasks
```

The work-efficiency of our fine-grained parallel Read-Tarjan algorithm can be demonstrated using the example given in Figure 4.2a. In this example, the threads of this algorithm independently explore four different path extensions $E_i = v_1 \to u_i \to v_2 \to v_0$, with $i \in \{1 \dots 4\}$. A thread exploring a path extension E_i invokes a DFS from v_2, which explores vertices $b_1, \dots, b_k$ at most once and fails to find any other path extension. Therefore, the amount of work the fine-grained parallel Read-Tarjan algorithm performs does not increase compared to its single-threaded execution.

Lemma 5. *The depth $T_\infty(n)$ of the fine-grained parallel Read-Tarjan algorithm is in $O(ne)$.*

Proof. In the worst case, a thread executing this algorithm creates a task for each vertex of its longest simple cycle, which has a length of at most n. Before invoking its first child task, a task executes a sequence of unsuccessful DFS invocations in $O(e)$ and a successful DFS invocation also in $O(e)$. Thus, the depth of this algorithm is $O(ne)$. □

The worst-case depth of our algorithm can be observed when this algorithm is executed on the graph given in Figure 4.1a. This graph has $c = 2^{n-2}$ cycles and the length of its longest cycle $v_0 \to \dots v_{n-1} \to v_0$ is n. The algorithm creates a task for each vertex of the cycle and performs a successful DFS in each such call, which leads to $T_\infty \in O(ne)$.

Theorem 11. *The fine-grained parallel Read-Tarjan algorithm is strongly scalable when* $\lim_{n\to\infty} c/n = \infty$.

Proof. Because the fine-grained parallel Read-Tarjan algorithm is work-efficient, we can apply Brent's rule [Bre74]:

$$\frac{T_1(n)}{p} \le T_p(n) \le \frac{T_1(n)}{p} + T_\infty(n). \tag{4.2}$$

Substituting $T_1(n)$ with $O(n+e+ec)$ and $T_\infty(n)$ with $O(ne)$ (see Lemma 5), for a positive constant C_0, it holds that

$$1 \Big/ \left(\frac{1}{p} + C_0\frac{n}{c}\right) < 1 \Big/ \left(\frac{1}{p} + \frac{T_\infty(n)}{T_1(n)}\right) \le \frac{T_1(n)}{T_p(n)} \le p. \tag{4.3}$$

Given that $\lim_{n\to\infty} c/n = \infty$, there exist $n_0 > 0, C_1 > 0$ such that if $n > n_0$, then $c/n > C_1 p$. Thus, for every $n > n_0$, it holds that $kp \leq \frac{T_1(n)}{T_p(n)} \leq p$, where $k = C_1/(C_0 + C_1) < 1$. As a result, $\frac{T_1(n)}{T_p(n)} = \Theta(p)$, which, based on Definition 5, completes the proof. □

As shown in Table 4.3, our fine-grained parallel Read-Tarjan algorithm has a higher depth than our fine-grained parallel Johnson algorithm, introduced in Section 4.3. Nevertheless, the former algorithm is strongly scalable when c grows superlinearly with n, whereas strong scalability cannot be guaranteed for the latter algorithm.

4.4.4 Summary

The work of our fine-grained parallel Read-Tarjan algorithm does not increase after fine-grained parallelisation. This parallel algorithm performs $W_p(n) \in O(n + e + ec)$ work: the same as the work performed by its serial version. Our optimisations introduced in Section 4.4.1 do not reduce the work $W_p(n)$ performed by our parallel algorithm in the worst case. However, these optimisations significantly improve its performance in practice (see Section 4.6.4). In addition, the synchronisation overheads of the fine-grained parallel Read-Tarjan algorithm are not as significant as those of the fine-grained Johnson algorithm because of its shorter critical sections. Furthermore, this algorithm is the only asymptotically-optimal parallel algorithm for cycle enumeration for which we are able to prove strong scalability.

4.5 Parallelising Constrained Cycle Search

This section describes the methods for adapting our parallel algorithms to search for simple cycles under various constraints. Because state-of-the-art algorithms for temporal and hop-constrained cycle enumeration are extensions of the Johnson algorithm [KC18; Pen+19], our parallelisation approach described in Section 4.3 is also applicable to these algorithms. In this section, we describe the changes to the fine-grained parallel Johnson algorithm needed for enumeration of temporal and hop-constrained cycles. We also introduce modifications to the cycle enumeration algorithms required for finding time-window-constrained cycles.

4.5.1 Time-Window Constraints

Cycle enumeration algorithms require minimal modifications to support time-window constraints. Such constraints restrict the search for simple, temporal, and hop-constrained cycles to those that occur within a time window of a given size δ, as illustrated in Figure 2.1. To find time-window-constrained cycles that start with an edge that has timestamp t_0, only the edges with timestamps that belong to the time window $[t_0 : t_0 + \delta]$ are visited. To avoid reporting the same cycle several times, another edge with the same timestamp t_0 is visited only if the source vertex of that edge has an ID that is smaller than the ID of the vertex from which the search for cycles was started. Overall, imposing time-window constraints reduces the number of cycles

Algorithm 14: CFGJ_copyOnSteal(d, T_1, T_2)

Input: d - the depth of the task executing this function
InOut: T_1 - the victim thread
T_2 - the stealing thread

```
1 Mutex_T1.lock();
2 {Π_T2, Blk_T2, PrevLocks_T2} = copy({Π_T1, Blk_T1, PrevLocks_T1}); ▷ Blk contains closing times or barriers
3 {Blist_T2} = copy({Blist_T1}); ▷ Blist is not used for hop-constrained cycles
4 Mutex_T1.unlock();
5 while |Π_T2| > d do ▷ Copy-on-steal with recursive unblocking
6 |   u = Π_T2.pop();
7 |   lock = PrevLocks_T2.pop();
8 |   RecursiveUnblock(u, lock, Blk_T2, Blist_T2);
```

discovered, which results in a more tractable problem.

A strongly-connected component (SCC) can be used to reduce the number of vertices visited during the search for time-window-constrained cycles. The search for cycles that start with the edge e can be limited to use only the vertices from the SCC that contains e [Joh75]. In the case of time-window-constrained cycles, we compute an SCC for e using only the edges with timestamps that belong to $[t_0 : t_0 + \delta]$, where t_0 is the timestamp of e. Because an SCC can be computed independently for each edge in $O(e)$ time [FHP00], our fine-grained parallel algorithms remain scalable.

4.5.2 Temporal Ordering Constraints

To efficiently enumerate temporal cycles, the 2SCENT algorithm [KC18] replaces the set of blocked vertices *Blk* in the Johnson algorithm with *closing times*. The closing time *ct* of a vertex v indicates that the outgoing temporal edges of v with a timestamp greater than or equal to *ct* cannot participate in a temporal cycle and are therefore blocked. Increasing the closing time of v to a new value ct' unblocks the blocked outgoing edges of v that have timestamps smaller than ct'. This operation triggers the recursive unblocking procedure that unblocks the incoming edges of v with a timestamp smaller than the maximal timestamp among the unblocked outgoing edges of v. This process is repeated for every vertex with unblocked outgoing edges.

Because the backtracking phase of 2SCENT is based on the Johnson algorithm, it can be parallelised using our fine-grained approach described in Section 4.3. For this purpose, we use our copy-on-steal mechanism with recursive unblocking, introduced in Section 4.3.2, which enables a thread to maintain its own set of data structures used for recursion tree pruning. However, this mechanism is not directly applicable in this case because the recursive unblocking procedure of 2SCENT requires the new closing time for a vertex as a parameter in addition to the vertex itself. For this reason, an additional data structure called *PrevLocks*

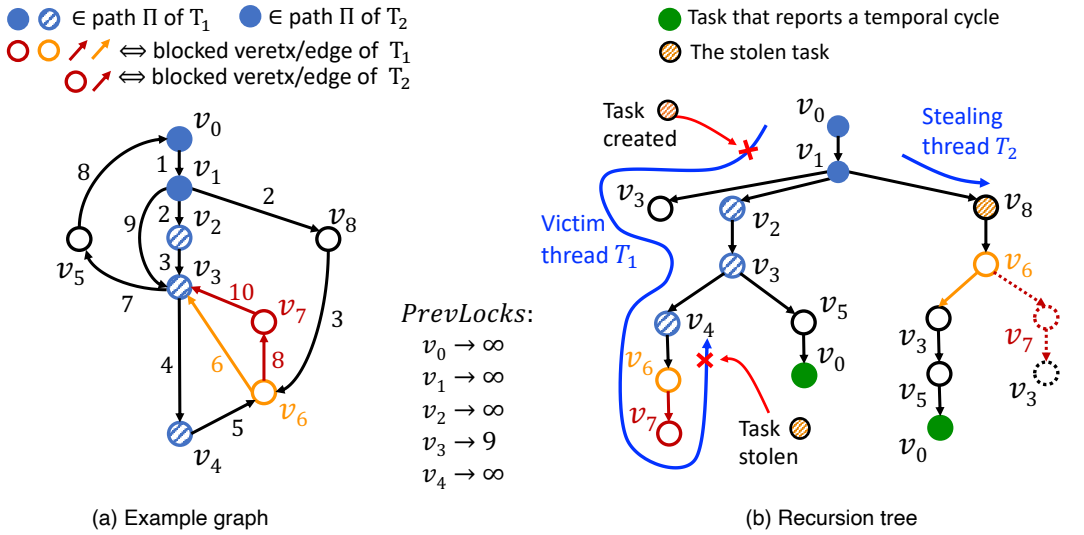


Figure 4.5: (a) An example graph and (b) the recursion tree of our fine-grained temporal Johnson algorithm when enumerating temporal cycles that start from v_0. The thread T_2 can avoid the dotted part of the tree by reusing the blocked edges $v_6 \rightarrow v_7$ and $v_7 \rightarrow v_3$ discovered by T_1.

is used alongside the current path Π that records the closing time that each vertex v had before it was added to Π. Copy-on-steal then performs the recursive unblocking procedure for each vertex v removed from Π using the original closing time of the vertex v obtained from *PrevLocks*, as shown in Algorithm 14. We refer to the resulting algorithm as the *fine-grained parallel temporal Johnson* algorithm.

The aforementioned modification to the copy-on-steal with recursive unblocking approach also enables a thread of our fine-grained parallel algorithm to reuse the edges blocked by another thread. This behaviour can be observed in the example shown in Figure 4.5, where the thread T_2 steals the task indicated in Figure 4.5b from the thread T_1. Copy-on-steal executed by T_2 invokes recursive unblocking that restores the closing time of v_3 to its original value of 9 obtained from *PrevLocks*. Note that this original closing time of v_3 was previously set by T_1 while exploring the path $v_0 \rightarrow v_1 \rightarrow v_3$. The recursive unblocking that T_2 invokes for v_3 unblocks only the edge $v_6 \rightarrow v_3$ because it is the only incoming edge of v_3 with a timestamp smaller than the closing time 9 of the vertex v_3. Without recording the previous closing times, T_2 could instead unblock all incoming edges of v_3 by invoking recursive unblocking for v_3 with a closing time ∞, which also unblocks the edges $v_6 \rightarrow v_7$ and $v_7 \rightarrow v_3$. However, because there is no temporal cycle that contains these two edges and starts with v_0, T_2 would unnecessarily visit them in this case. Thus, restoring the closing time of v_3 to its original value 9 prevents T_2 from performing this redundant work.

We also adapt the Read-Tarjan algorithm and its fine-grained and coarse-grained versions to enumerate temporal cycles using closing times. The necessary changes to the algorithm are trivial, and we omit discussing them for brevity.

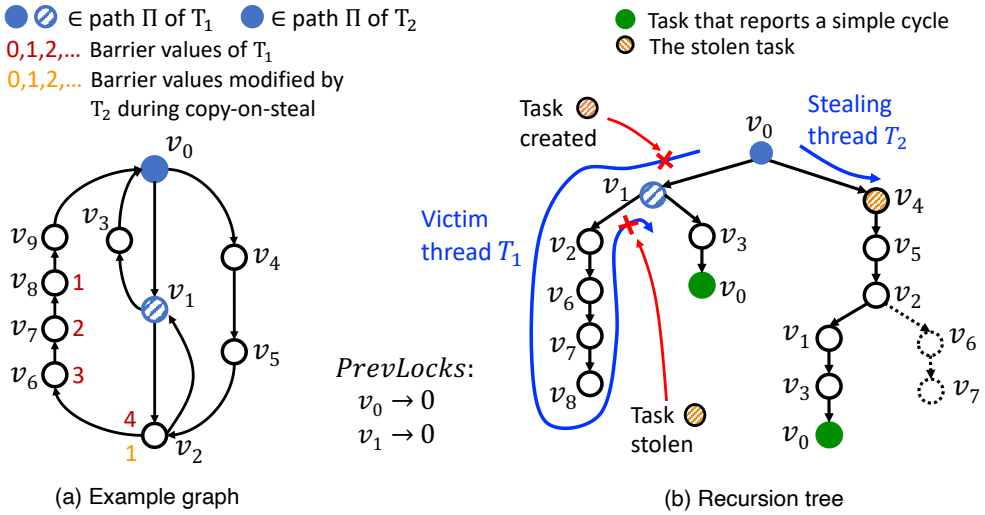


Figure 4.6: (a) An example graph and (b) the recursion tree of our fine-grained hop-constrained Johnson algorithm when enumerating cycles of length $L = 6$ that start from v_0. Barrier values of unmarked vertices are 0. Copy-on-steal enables the thread T_2 to reuse barriers discovered by the thread T_1 and to avoid exploring the dotted part of the tree.

To reduce the number of vertices visited during the search for temporal cycles, we use a method similar to the SCC-based technique discussed in Section 4.5.1. Instead of computing an SCC for each edge e, we compute a *cycle-union* that represents an intersection of temporal ancestors and temporal descendants of e. The temporal descendants and the temporal ancestors of e are the vertices that belong to the temporal paths in which e is the first edge and the last edge, respectively. Defined as such, a cycle-union contains only the vertices that participate in temporal cycles that have e as their starting edge. Thus, the search for temporal cycles that start with e can be limited to only those vertices.

4.5.3 Hop Constraints

An efficient algorithm for enumerating hop-constrained cycles and paths, called BC-DFS [Pen+19], replaces the set of blocked vertices *Blk* in the Johnson algorithm with *barriers*. A barrier value *bar* of a vertex v indicates that the starting vertex v_0 of a cycle cannot be reached within *bar* hops from v. As a result, v is blocked if the length of the current path Π when the algorithm attempts to visit v is greater than or equal to $L - bar$, where L is the hop constraint. BC-DFS modifies the recursive unblocking of the Johnson algorithm to reduce the barrier *bar* of v to a specified value $bar' < bar$. This procedure also sets the barrier of any vertex u that can reach v in k hops to $bar' + k$ if the previous barrier of u was greater than $bar' + k$. Maintaining barriers in such a way minimises redundant vertex visits when searching for hop-constrained cycles.

To parallelise BC-DFS in a fine-grained manner, we use the same technique as that used for fine-grained parallelisation of the Johnson algorithm (Section 4.3) and the 2SCENT algorithm (Section 4.5.2). In this case, threads exploring a recursion tree of BC-DFS maintain separate

Table 4.4: Hardware platforms used in the cycle enumeration experiments. Here, P, C/P, and T/C represent the number of processors, the number of cores per processor, and the number of hardware threads per core, respectively.

platform	Intel KNL [Sod15]	Intel Xeon Skylake [Goo22]
P × C/P × T/C	4 × 64 × 4	5 × 48 × 2
Total no. threads	1024	480
Frequency	1.3 GHz	2 GHz
Memory per proc.	110 GB	360 GB
L1d/L2/L3 cache	32 KB/512 KB/none	32 KB/1 MB/38.5 MB

data structures, such as the current path Π and barrier values for each vertex, and use the copy-on-steal with the recursive unblocking approach to copy these data structures among threads. Similarly to our algorithm from Section 4.5.2, each thread also maintains a data structure *PrevLocks* that records the original barrier value of each vertex v from Π. When a thread steals a task, it performs a recursive unblocking procedure for each vertex v removed from Π using its original barrier value obtained from *PrevLocks*, as shown in Algorithm 14. This procedure reduces the barrier value of the vertices that can reach v, enabling the stealing thread to visit those vertices. We refer to the resulting algorithm as the *fine-grained parallel hop-constrained Johnson* algorithm.

The modified copy-on-steal with recursive unblocking approach given in Algorithm 14 enables a stealing thread of the aforementioned fine-grained parallel algorithm to reuse barriers discovered by other threads. This behaviour can be observed in the example given in Figure 4.6. In that example, the thread T_1 first visits the vertices v_2, v_6, v_7, v_8 and sets the barrier value of each visited vertex to $L - |\Pi| + 1$ (values in red shown in Figure 4.6a) because it was not able to find a cycle of length $L = 6$ [Pen+19]. Here, $|\Pi|$ denotes the length of Π at the moment of exploration of each vertex. When the thread T_2 steals the task indicated in Figure 4.6b from T_1, the copy-on-steal mechanism executed by T_2 performs a recursive unblocking of the vertex v_1 using the original barrier value 0 of v_1 obtained from *PrevLocks*. This recursive unblocking reduces the barrier value of v_2 from 4 to 1, which enables T_2 to find the cycle that contains v_2. The barrier values of the vertices v_6, v_7, and v_8 are not modified, and, thus, the thread T_2 avoids visiting these vertices unnecessarily.

4.5.4 Summary

In this section, we described a method to adapt the cycle enumeration algorithms, such as our fine-grained algorithms introduced in Sections 4.3 and 4.4, to search for cycles under time window constraints. In addition, we introduced a modified version of our copy-on-steal with recursive unblocking approach, introduced in Section 4.3, that supports fine-grained parallelisation of temporal and hop-constrained cycle enumeration algorithms [KC18; Pen+19] derived from the Johnson algorithm. As a result, our fine-grained parallel algorithms can enumerate cycles under time-window, temporal, and hop constraints.

Table 4.5: Temporal graphs used in the cycle enumeration experiments. Time span refers to the difference between the maximum and minimum timestamps in a graph.

graph	abbr.	No. vertices	No. edges	Time span [days]
bitcoinalpha	BA	3.3 k	24 k	1901
bitcoinotc	BO	4.8 k	36 k	1903
CollegeMsg	CO	1.3 k	60 k	193
email-Eu-core	EM	824	332 k	803
mathoverflow	MO	16 k	390 k	2350
transactions	TR	83 k	530 k	1803
higgs-activity	HG	278 k	555 k	6
askubuntu	AU	102 k	727 k	2613
superuser	SU	138 k	1.1 M	2773
wiki-talk	WT	140 k	6.1 M	2277
friends2008	FR	481 k	12 M	1826
wiki-dynamic-nl	NL	1 M	20 M	3602
messages	MS	313 k	26 M	1880
AML-Data	AML	10 M	34 M	30
stackoverflow	SO	2.0 M	48 M	2774

4.6 Experimental Evaluation

This section evaluates the performance of our fine-grained parallel algorithms for simple, temporal, and hop-constrained cycle enumeration. As Table 4.2 shows, we are the only ones to offer fine-grained parallel versions of the asymptotically-optimal cycle enumeration algorithms, such as the Johnson and the Read-Tarjan algorithms. However, the methods covered in Table 4.2 can be parallelised using the coarse-grained approach covered in Section 4.2. Thus, we use the coarse-grained approach as our main comparison point.

The experiments are performed using two different clusters: Intel KNL [Sod15] and Intel Xeon Skylake [Goo22]. The details of these two clusters are given in Table 4.4. We developed our code on the Intel KNL cluster and ran most of the analyses there; yet, for completeness, we ran the comparisons to competing implementations also on the Intel Xeon Skylake cluster available in Google Cloud's Compute Engine [Goo22]. Scalability experiments are conducted on the Intel KNL cluster. In these experiments, the data points that use 64 threads or less were executed on a single Intel KNL processor; two processors were used to execute the data points that use 128 threads; and all four processors were used otherwise. Furthermore, we use more than one thread per core only if the number of threads used is greater than 256.

We use the *Threading Building Blocks* (TBB) [Kuk07] library to parallelise the algorithms on a single processor. We distribute the execution of the algorithms across multiple processors using the Message Passing Interface (MPI) [COR93]. When using distributed execution, each processor stores a copy of the input graph in its main memory and searches for cycles starting from a different set of graph edges. The starting edges are divided among the processors such that when the edges are ordered in the ascending order of their timestamps, k consecutive edges in that order are assigned to k different processors. Each processor then uses its own dynamic scheduler to balance the workload across its hardware threads. In this setup, work-

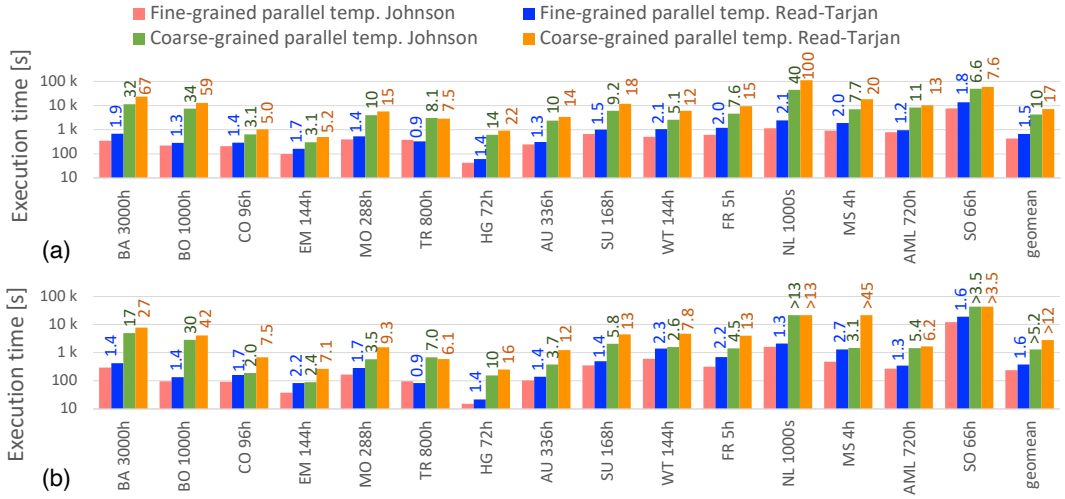


Figure 4.7: Performance of parallel algorithms for temporal cycle enumeration on (a) the Intel KNL cluster using 1024 threads and (b) the Intel Xeon Skylake cluster using 480 threads. The numbers above the bars show the execution time of each algorithm relative to that of our fine-grained parallel temporal Johnson for the same benchmark.

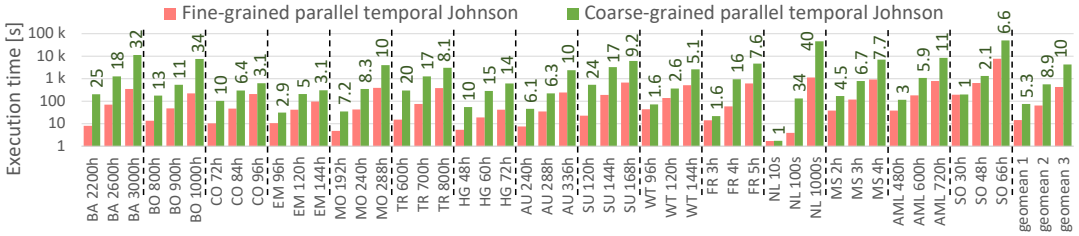


Figure 4.8: Larger time windows increase the performance gap between the algorithms. The algorithms are executed on the Intel KNL cluster using 1024 threads. The numbers above the bars show the execution times of the coarse-grained algorithm relative to that of the fine-grained algorithm.

load imbalance across processors may still occur, but its impact is limited in our experiments because we use at most five processors.

We perform the experiments using the graphs listed in Table 4.5. The TR, FR, and MS graphs are from *Harvard Dataverse* [JMB17], the NL graph is from *Konect* [Kun13], the AML graph is from the *AML-Data* repository [Alt21], and the rest are from *SNAP* [LK14]. To make cycle enumeration problems tractable, we use time-window constraints in all of our experiments. The time window sizes used in our experiments are given in the figures next to the graph names. We stop the execution of an algorithm if it takes more than $24h$ on the Intel KNL cluster or more than $6h$ on the Intel Xeon Skylake cluster.

4.6.1 Temporal Cycle Enumeration

The goal of a temporal cycle enumeration problem is to find all simple cycles with edges ordered in time. Here, we evaluate the performance of our fine-grained parallel algorithms

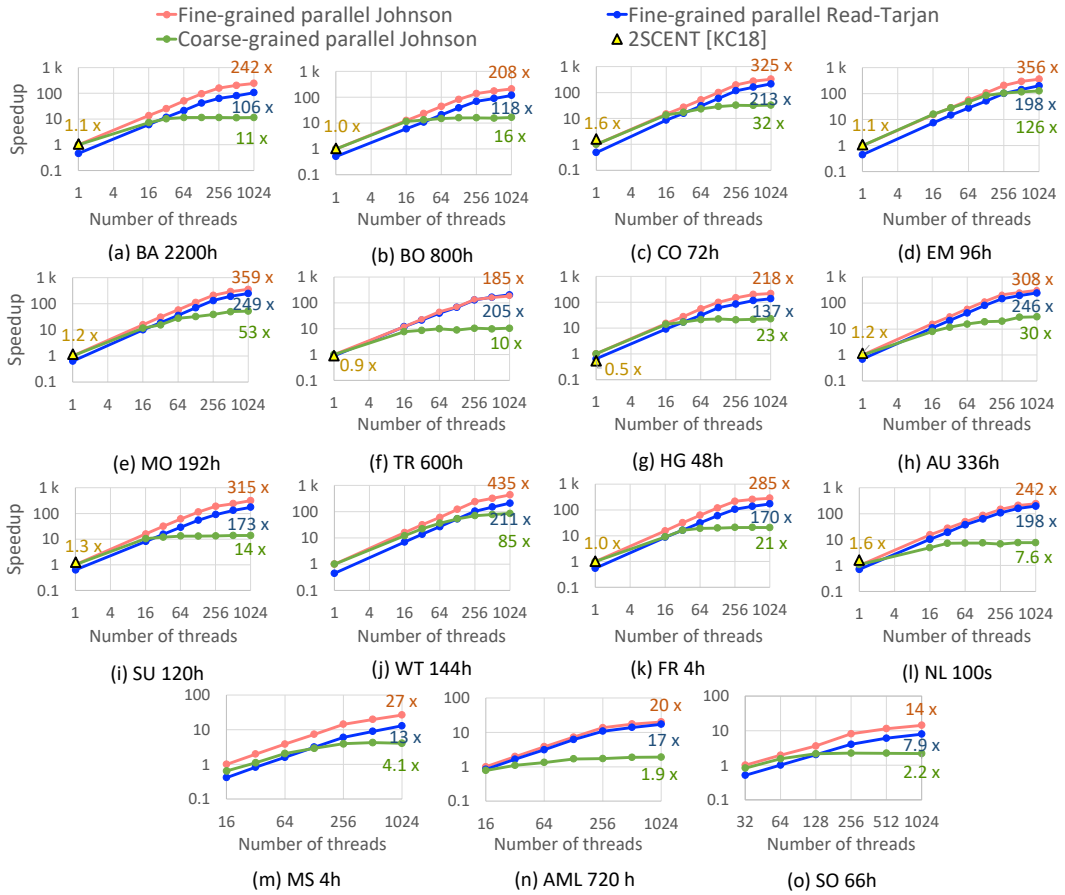


Figure 4.9: Scalability evaluation of parallel temporal cycle enumeration algorithms executed on the Intel KNL cluster. The baseline is our fine-grained parallel temporal Johnson algorithm. The relative performance of 2SCENT [KC18] is shown when it completes in 24 hours. Note that the 2SCENT implementation is single-threaded and the single-threaded execution results are not available for all graphs.

for this problem introduced in Section 4.5.2. Our main comparison points are the coarse-grained parallel versions of the temporal Johnson and temporal Read-Tarjan algorithms. We refer to the backtracking phase of the state-of-the-art 2SCENT algorithm [KC18] for temporal cycle enumeration as the temporal Johnson algorithm and parallelise it in a coarse-grained manner for the experiments. We do not parallelise the entire 2SCENT algorithm because the preprocessing phase of 2SCENT is strictly sequential and has a time complexity in the order of the complexity of its backtracking phase. We also provide direct comparisons with the 2SCENT algorithm.

Figure 4.7 shows that our fine-grained parallel algorithms achieve an order of magnitude speedup compared to the coarse-grained algorithms on the Intel KNL cluster. For the NL graph, this speedup reaches up to 40×. Because the Intel Xeon Skylake cluster contains fewer physical cores than the Intel KNL cluster, the speedup between our fine-grained and the coarse-grained parallel Johnson algorithms is smaller on the former cluster. As can be

observed in Figure 4.8, this speedup increases as we increase the time window size used in the algorithms. Note that enumerating cycles in larger time windows is more challenging because larger time windows contain a larger number of cycles.

The scalability evaluation of the parallel temporal cycle enumeration algorithms is given in Figure 4.9. We also report the performance of the sequential 2SCENT algorithm in the same figure. The performance of our fine-grained parallel algorithms improves linearly until 256 threads, after which it becomes sublinear due to simultaneous multithreading. As a result, our fine-grained versions of the Johnson and the Read-Tarjan algorithms reach 435× and 470× speedups, respectively, compared to their serial versions. Additionally, when using 1024 threads, our fine-grained Johnson algorithm is on average 260× faster than 2SCENT when 2SCENT completes in 24 hours. On the other hand, the coarse-grained Johnson algorithm does not scale as well as the fine-grained algorithms. As a result, the performance gap between the fine-grained and the coarse-grained algorithms increases as we increase the number of threads.

Overall, the fastest algorithm for temporal cycle enumeration that we tested is our fine-grained Johnson algorithm, which is, on average, 60% faster than our fine-grained Read-Tarjan algorithm. When using 1024 threads, both fine-grained algorithms are an order of magnitude faster than their coarse-grained counterparts. Moreover, our fine-grained parallel algorithms, executed on the Intel KNL cluster using 1024 threads, are two orders of magnitude faster than the state-of-the-art algorithm 2SCENT [KC18].

4.6.2 Hop-Constrained Cycle Enumeration

In hop-constrained cycle enumeration, we search for all simple cycles in a graph that are shorter than the specified hop constraint. We here compare our fine-grained parallel hop-constrained Johnson algorithm, introduced in Section 4.5.3, with the state-of-the-art algorithms BC-DFS and JOIN [Pen+19] for this problem. For this evaluation, we parallelised BC-DFS and JOIN in the coarse-grained manner. Because adapting the Read-Tarjan algorithm to enumerate hop-constrained cycles is not trivial, we do not report the performance of the fine-grained and coarse-grained versions of this algorithm. We also omit the performance results for the MS graph because our fine-grained algorithm did not finish under $12h$ when using the smallest time window size.

Figure 4.10 shows that our fine-grained parallel algorithm is, on average, more than 10× faster than the coarse-grained parallel BC-DFS algorithm for the two largest hop constraints tested. When using the hop-constraint that is less than or equal to ten, the coarse-grained parallelisation approach is able to achieve workload balance across cores, and, thus, the performance of this approach is similar to that of our fine-grained approach in this case. As we increase the hop constraint, the probability of encountering deeper recursion trees also increases. Exploring such trees using the coarse-grained approach leads to workload imbalance (see Section 4.2). Our fine-grained algorithm is designed to resolve this problem by exploring a

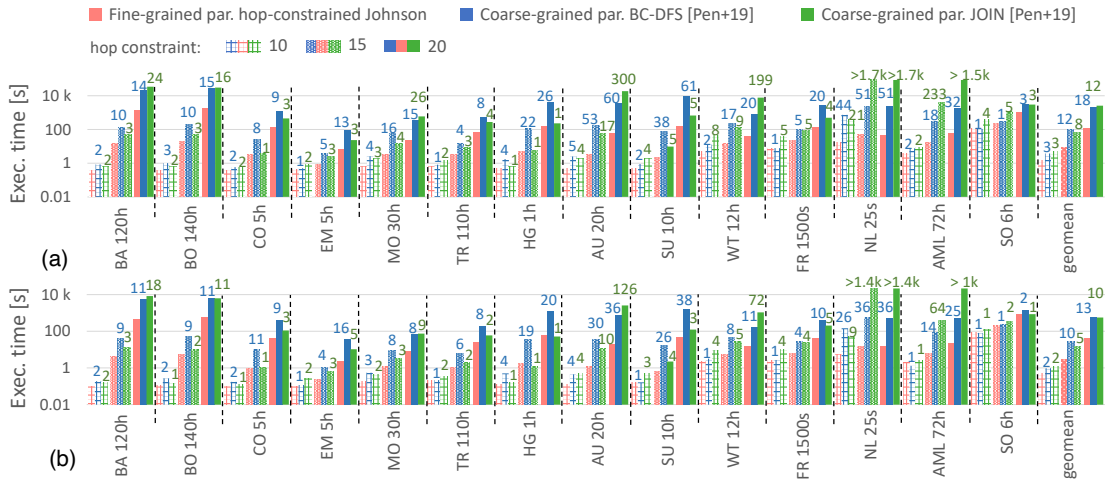


Figure 4.10: Performance of parallel algorithms for hop-constrained simple cycle enumeration on (a) the Intel KNL cluster using 1024 threads and (b) the Intel Xeon Skylake cluster using 480 threads. The numbers above the bars show the execution time of the coarse-grained parallel algorithms relative to that of our fine-grained parallel algorithm. Larger hop constraints increase the performance gap between the two algorithms.

recursion tree using several threads. Therefore, increasing the hop constraint increases the speedup of our fine-grained algorithm with respect to the coarse-grained algorithm.

When the hop constraint is set to 20, our fine-grained parallel algorithm is, on average, 10× faster than the coarse-grained parallel JOIN algorithm, as shown in Figure 4.10. Although the latter algorithm can be competitive with our fine-grained algorithm, it can also suffer from long execution times, such as in the cases of the AU, NL, and AML graphs. The reason for these long execution times is the fact that the JOIN algorithm might temporarily construct many non-simple cycles while searching for simple cycles. Because this algorithm constructs cycles by combining simple paths, it is not guaranteed that each combination results in a simple cycle. The overhead of combining paths can dominate the execution time of JOIN if this algorithm constructs orders of magnitude more non-simple cycles than simple cycles. For instance, this situation occurs in the case of AU and hop constraint of 20, where JOIN discovers 600× more non-simple cycles than simple cycles. As a result, the speedup of our fine-grained algorithm compared to the coarse-grained JOIN algorithm can reach up to three orders of magnitude.

Figure 4.11 shows that the speedup of our fine-grained parallel Johnson algorithm with respect to the coarse-grained parallel BC-DFS can be increased by using more threads. The performance of our fine-grained parallel algorithm scales linearly with the number of threads, whereas the scaling of the coarse-grained parallel BC-DFS eventually slows down. Thus, in addition to being, on average, an order of magnitude faster than the coarse-grained parallel BC-DFS, our fine-grained algorithm is also more scalable.

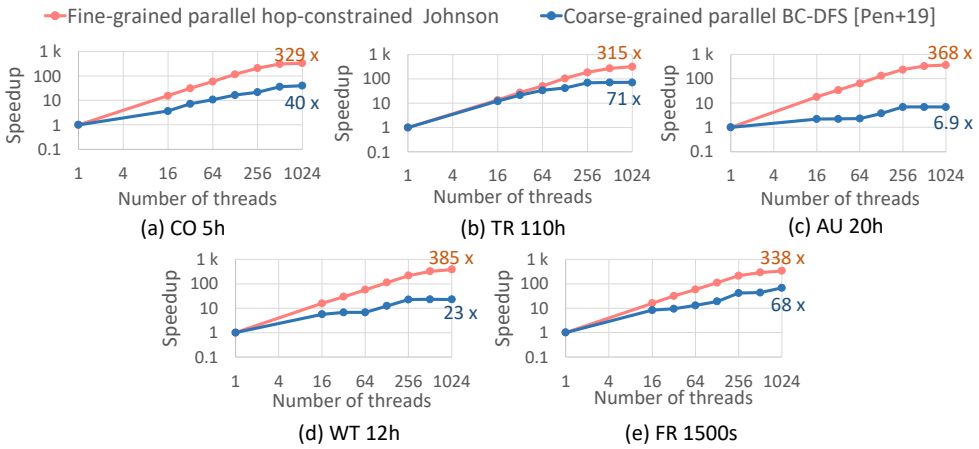


Figure 4.11: Scalability evaluation of parallel hop-constrained cycle enumeration algorithms executed on the Intel KNL cluster using the hop constraint of 15. The speedup values are relative to the single-threaded execution of BC-DFS. Evaluation on other graphs is omitted for brevity.

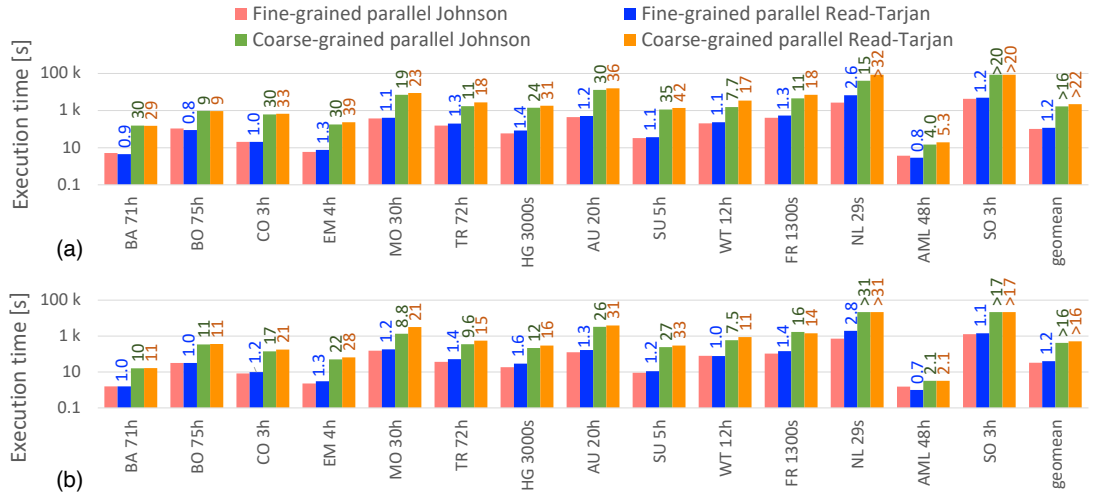


Figure 4.12: Performance of parallel algorithms for simple cycle enumeration on (a) the Intel KNL cluster using 1024 threads and (b) the Intel Xeon Skylake cluster using 480 threads. The numbers above the bars show the execution time of each algorithm relative to that of our fine-grained parallel Johnson algorithm for the same benchmark.

4.6.3 Simple Cycle Enumeration

Here, we evaluate our fine-grained parallel algorithms for simple cycle enumeration. The computational complexity of simple cycle enumeration is higher than the complexity of temporal and hop-constrained cycle enumeration because simple cycle enumeration does not impose temporal ordering or hop constraints. The only constraint we impose is the time-window constraint. Because the complexity of enumerating simple cycles is higher, we use smaller time windows compared to the cases of temporal and hop-constrained cycle enumeration. We use the coarse-grained parallel versions of the Johnson and the Read-Tarjan

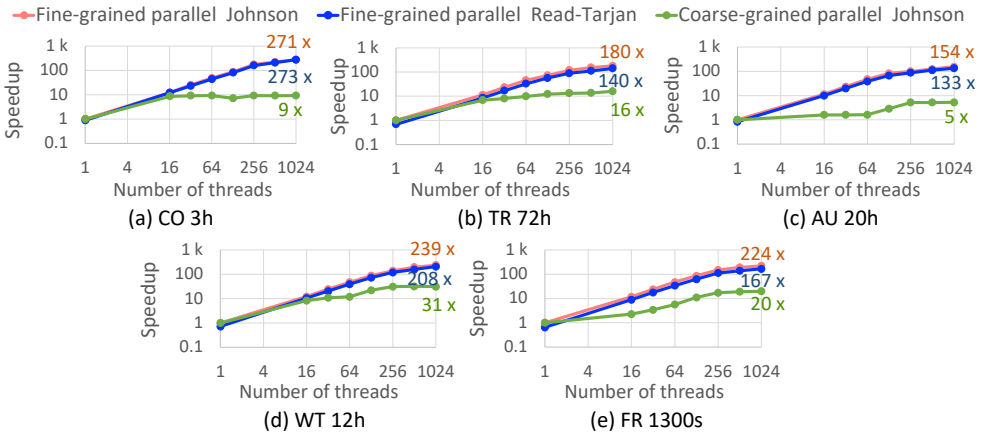


Figure 4.13: Scalability evaluation of parallel simple cycle enumeration algorithms executed on the Intel KNL cluster. The speedup values are relative to the single-threaded execution of the Johnson algorithm. Evaluation on other graphs is omitted for brevity.

algorithms as our main comparison points. We do not report the results for the MS graph because our algorithms did not finish in $12h$ even if we set the time window to one second.

As we can see in Figure 4.12, our fine-grained parallel algorithms show an order of magnitude average speedup compared to coarse-grained parallel algorithms on two different platforms. The reason for this speedup is better scalability of our fine-grained algorithms, which we demonstrate in Figure 4.13. Similarly to the cases of temporal and hop-constrained cycle enumeration (see Figs. 4.9 and 4.11), our fine-grained parallel algorithms scale linearly with the number of physical cores used whereas the coarse-grained parallel Johnson algorithm does not scale as well. Thus, the speedup between the fine-grained and the coarse-grained algorithms increases by utilising more threads.

The synchronisation overheads caused by recursive unblocking of our fine-grained parallel Johnson algorithm (see Section 4.3.2) are visible only in the case of AML. In this case, the fine-grained parallel Johnson algorithm performs 60% fewer edge visits than the fine-grained parallel Read-Tarjan; however, it is 25% slower. These synchronisation overheads can be explained by a very low cycle-to-vertex ratio. Because a vertex is blocked if it cannot take part in a cycle, the probability of a vertex being blocked is higher when the cycle-to-vertex ratio is lower. In consequence, more vertices are unblocked during the recursive unblocking of the fine-grained parallel Johnson algorithm, leading to longer critical sections and more contention on the locks. Nevertheless, our fine-grained parallel Johnson algorithm achieves a good trade-off between pruning efficiency and lock contention in most cases.

Overall, our fine-grained parallel Johnson and fine-grained parallel Read-Tarjan algorithms have comparable performance, as shown in Figure 4.12. Although the former algorithm is slightly faster, it can suffer from synchronisation overheads in some cases. Nevertheless, both parallel algorithms achieve almost linear scaling with the number of physical cores used and achieve, on average, more than 10× speedup with respect to coarse-grained parallel

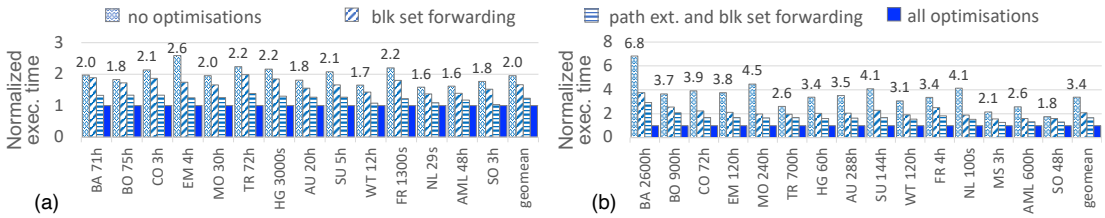


Figure 4.14: Effect of the pruning improvements to our fine-grained parallel Read-Tarjan algorithm for (a) simple and (b) temporal cycle enumeration. Execution times are normalised to the case that includes all optimisations. Our optimisations accelerate this algorithm by up to 6.8×.

versions of the algorithms. These conclusions also hold in the cases of temporal and hop-constrained cycle enumeration.

4.6.4 Improvements to the Read-Tarjan Algorithm

Figure 4.14 shows the effect of our pruning improvements, introduced in Section 4.4.1, on the performance of our fine-grained Read-Tarjan algorithm. The experiments are performed using a single Intel KNL processor using 256 threads. Note that using one processor instead of the entire cluster results in longer execution times, but it enables us to eliminate the effect of workload imbalance across processors in this experiment. The execution time of the fine-grained parallel Read-Tarjan algorithm decreases after activating each optimisation because fewer redundant vertex and edge visits are performed during the execution of this algorithm. When all optimisations are enabled, the average speedup of our algorithm for simple cycle enumeration compared to its unoptimised version is 2×. In the case of temporal cycle enumeration, the average speedup increases to 3.4×. As a result, our pruning improvements enable the fine-grained parallel Read-Tarjan algorithm to be competitive with the fine-grained parallel Johnson algorithm.

4.7 Related Work

Simple cycle enumeration algorithms. Enumeration of simple cycles of graphs is a classical computer science problem [Tie70; Tar73; Joh75; RT75; MD76; SL76; Gro16; Wei72; LT82; Bir+13; AR16]. The backtracking-based algorithms by Johnson [Joh75], Read and Tarjan [RT75], and Szwarcfiter and Lauer [SL76] achieve the lowest time complexity bounds for enumerating simple cycles in directed graphs. These algorithms implement advanced recursion tree pruning techniques to improve on the brute-force Tiernan algorithm [Tie70]. Section 2.4.2 covers such pruning techniques in further detail. A cycle enumeration algorithm that is asymptotically faster than the aforementioned algorithms [Joh75; RT75; SL76] has been proposed in Birmelé et al. [Bir+13], however, it is applicable only to undirected graphs. Simple cycles can also be enumerated by computing the powers of the adjacency matrix [Dan68; Kam67; Pon66] or by using circuit vector space algorithms [MD76; Gib69; Wel65], but the complexity of such

approaches grows exponentially with the size of the cycles or the size of the input graphs.

Time-window, temporal ordering, and hop constraints. It is common to search for cycles under some additional constraints. For instance, in temporal graphs, it is common to search for cycles within a sliding time window, such as in Kumar and Calders [KC18] and Qiu et al [Qiu+18]. In addition, temporal ordering constraints can be imposed when searching for cycles in temporal graphs, such as in Kumar and Calders [KC18]. Furthermore, the maximum number of hops in cycles or paths can be constrained, such as in Gupta and Suzumura [GS21] and Peng et al. [Pen+19]. Note that hop-constrained simple cycles can also be enumerated using incremental algorithms, such as in Qiu et al. [Qiu+18]. However, this algorithm is based on the brute-force Tiernan algorithm [Tie70], which makes it slower than nonincremental algorithms that use recursion tree pruning techniques [Pen+19]. Additionally, because incremental algorithms maintain auxiliary data structures, such as paths, to be able to construct cycles incrementally, they are not as memory-efficient as nonincremental algorithms [Pen+19]. Table 4.2 offers comparisons between the capabilities of these methods and ours.

Parallel and distributed algorithms for cycle enumeration. Cui et al. [Cui+17] proposed a multi-threaded algorithm for detecting and removing simple cycles of a directed graph. The algorithm divides the graph into its strongly-connected components and each thread performs a depth-first search on a different component to find cycles. However, sizes of the strongly-connected components in real-world graphs can vary significantly [Meu+14], which leads to a workload imbalance. Rocha and Thatte [RT15] proposed a distributed algorithm for simple cycle enumeration based on the bulk-synchronous parallel model [Val90], but it searches for cycles in a brute-force manner. Qing et al. [Qin+20] introduced a parallel algorithm for finding length-constrained simple cycles. It is the only other fine-grained parallel algorithm we are aware of in the sense that it can search for cycles starting from the same vertex in parallel. However, the way this algorithm searches for cycles is similar to the way the brute-force Tiernan algorithm [Tie70] works. To our knowledge, we are the first ones to introduce fine-grained parallel versions of asymptotically-optimal simple cycle enumeration algorithms, which do not rely on a brute-force search, as we show in Table 4.2. Distributed algorithms for detecting the presence of cycles in graphs readily exist [Bad99; Oli+18; FO19]. However, our focus is on discovering all simple cycles of a graph rather than detecting whether a graph has a cycle or not.

4.8 Conclusions

This work presented in this chapter has made three contributions to the area of parallel cycle enumeration. First, we have introduced scalable fine-grained parallel versions of the state-of-the-art Johnson and Read-Tarjan algorithms for enumerating simple cycles. In particular, we have shown that the novel fine-grained parallel approach we contributed for parallelising the Johnson algorithm can be adapted to support the enumeration of temporal and hop-constrained cycles as well. Our fine-grained parallel algorithms for enumerating the afore-

mentioned types of cycles achieve a near-linear performance scaling on a compute cluster with a total number of 256 CPU cores that can execute 1024 simultaneous software threads.

Secondly, we have shown that our fine-grained parallel cycle enumeration algorithms are scalable both in theory and in practice. In contrast, their coarse-grained parallel versions do not share this property. When using 1024 software threads, our fine-grained parallel algorithms are on average an order of magnitude faster than their coarse-grained counterparts. In addition, the performance gap between the fine-grained and coarse-grained parallel algorithms widens as we use more physical CPU cores. This performance gap also widens when increasing the time window in the case of temporal cycle enumeration and when increasing the hop constraint in the case of hop-constrained cycle enumeration.

Thirdly, we have shown that, whereas our fine-grained parallel Read-Tarjan algorithm is work efficient, our fine-grained parallel Johnson algorithm is not. In general, the former is competitive against the latter because of the new pruning methods we introduced, yet the latter outperforms the former in most experiments. In some rare cases, our fine-grained parallel Johnson algorithm can suffer from synchronisation overheads. In such cases, our fine-grained parallel Read-Tarjan algorithm offers a more scalable alternative.

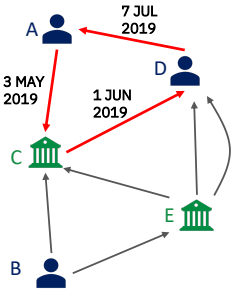
5 Graph Feature Extraction for Financial Crime Detection

In this chapter, we focus on analysing the financial transaction data with the goal of detecting transactions associated with financial crime. In a typical banking use-case, financial transactions take place dynamically, and a transaction processing system receives a list of financial transactions as input, often in tabular format, as shown in Figure 5.1a, where each transaction is represented by a row [IBM23c]. By treating each transaction as an edge and each account as a vertex, this tabular data can be transformed into graph format, as illustrated in Figure 5.1b. In such a financial transaction graph, the existence of various types of subgraph pattern can be associated with financial crime, as discussed in Section 1.1. For instance, cycles and scatter-gather patterns, illustrated in Figure 1.2, could indicate money laundering, stock market manipulation, or another financial fraud scheme [NKL21]. Therefore, extracting these subgraphs using fast enumeration algorithms, such as our parallel cycle enumeration algorithm introduced in Chapter 4, could accelerate the detection of financial crime.

This chapter introduces *Graph Feature Preprocessor*, a software library for detecting well-known money laundering and fraud patterns in financial transaction graphs, which we use to produce a rich set of account and transaction features. Our library efficiently extracts subgraph patterns using parallel algorithms, such as our parallel cycle enumeration algorithm presented in Chapter 4, from the incoming transaction stream. These subgraph patterns are encoded into the feature vectors of transactions that participate in those patterns, which can then be used to improve the accuracy of machine learning models that perform financial crime detection. In addition, to further enrich the feature set of transactions, our library computes various statistical properties of the graph vertices involved in the transactions. Furthermore, to enable real-time ingestion of transactions in small batches, our library maintains an in-memory directed graph data structure that supports fast dynamic updates. Because, in financial transaction graphs, several different transactions between the same two accounts can take place at different times, as illustrated in Figure 5.1, our in-memory directed graph data structure is implemented as a multigraph. Our Graph Feature Preprocessor is publicly available on PyPI [IBM22a; IBM22c; IBM22b] and is offered as part of IBM Cloud Pak for Data (CP4D 4.6.3) [IBM23a].

Trans. ID	Timestamp	Source bank ID	Source Account	Target bank ID	Target Account	Amount	Currency	Payment type
0	3 MAY 2019 12:45	1	A	1	C	1400	USD	Cheque
1	15 MAY 2019 07:34	2	B	1	C	710	EUR	ACH
2	18 MAY 2019 16:55	3	E	1	C	950	USD	Credit card
3	1 JUN 2019 10:06	1	C	3	D	1200	CHF	Wire
4	27 JUN 2019 13:18	3	E	3	D	2300	EUR	Credit card
5	7 JUL 2019 11:14	3	D	1	A	1100	USD	Credit card
6	14 JUL 2019 09:37	2	B	3	E	650	USD	ACH
7	20 JUL 2019 14:02	3	E	3	D	2500	USD	Wire

(a)



(b)

Figure 5.1: Financial transactions in (a) tabular format and in (b) graph format. The highlighted transactions form a money laundering cycle.

To evaluate our Graph Feature Preprocessor library, we have developed a graph machine learning (Graph ML) pipeline for monitoring financial transaction graphs. The pipeline uses our library to generate rich feature vectors for financial transactions and tree-based machine learning models to predict suspicious transactions. The dynamic in-memory multigraph our Graph Feature Preprocessor maintains enables the pipeline to dynamically process transactions in small batches. We have evaluated our pipeline using highly imbalanced anti-money laundering (AML) and phishing detection datasets, in which only a small fraction of the transactions are illicit, making the learning extremely challenging. When using an AML dataset with 100 million transactions, our graph-based features increased the minority-class F1 score from 9% to 73% while achieving a latency of 1 ms for batches of 128 transactions, culminating in throughput rates beyond 100′000 transactions per second using only 6 CPU cores. Additionally, our experiments on a real-life phishing account detection dataset extracted from Ethereum demonstrate the general applicability of our graph feature extraction library.

The rest of this chapter is organised as follows. The high-level overview of our solution for the detection of financial transactions associated with financial crime is given in Section 5.1. The details of our graph-based feature extraction library including its interface and feature encoding are given in Section 5.2. Our dynamic in-memory directed multigraph data structure, used by our feature extraction library, is introduced in Section 5.3. Section 5.4 describes the details of our graph machine learning pipeline. Section 5.5 presents the experimental evaluation of our solution. Section 5.6 discusses related work. This chapter is concluded in Section 5.7.

5.1 Overview of the Solution

To process data in tabular format, such as the financial transactions shown in Figure 5.1a, tree-based machine learning models [Ke+17; CG16] are often used [GOV22]. However, such models cannot take into account the underlying graph structure and cannot discover complex graph patterns in financial transaction graphs that could be associated with money laundering (see Figure 1.2). Furthermore, financial transactions are usually associated with a limited set of basic features (columns in Figure 5.1a) [IBM23c], which do not provide sufficient information

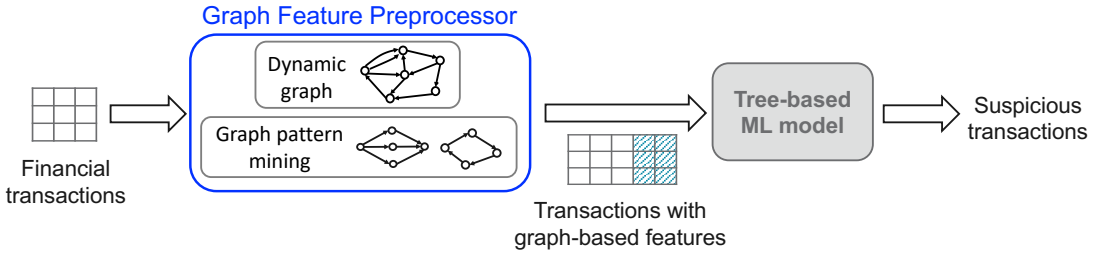


Figure 5.2: The overview of our graph machine learning pipeline for the detection of suspicious financial transactions. This setup uses our *Graph Feature Preprocessor* library to produce a rich set of graph-based features and a pre-trained machine learning model that uses these features to detect suspicious transactions.

to tree-based models for detecting transactions associated with financial crime. As a result, it is challenging to detect transactions associated with financial crime using these models.

To overcome the aforementioned limitations, we propose a solution shown in Figure 5.2. In this solution, we use our *Graph Feature Preprocessor* library to produce a rich set of graph-based features for financial transactions. Our library searches for well-known financial crime patterns, such as money laundering cycles and scatter-gather patterns (see Figure 1.2), and encodes these graph patterns into additional columns of the financial transaction table. The transaction table enriched with the graph-based features is then forwarded to a pre-trained tree-based machine learning model that performs the classification of financial transactions. As a result, the machine learning model is provided with additional transaction features extracted from the financial transaction graph, which facilitates the detection of transactions associated with financial crime.

Another benefit of our setup, shown in Figure 5.2, is that it can process transactions in small batches with high throughput. Upon receiving a batch of transactions, our *Graph Feature Preprocessor* library first updates an in-memory dynamic graph data structure with the edges that represent the transactions from the input batch. Then, for each edge e added to the graph, our library searches for the graph patterns that contain that edge e and encodes those graph patterns into the features of the transaction associated with e. Transaction processing can be parallelised by adopting the coarse-grained parallel approach, in which a graph pattern search for each edge e of a batch is performed by a different thread. However, as illustrated in Section 1.2 and Chapter 4, using the coarse-grained approach might result in a suboptimal solution due to the potential workload imbalance across threads. Furthermore, the parallelism of such approach is limited by the size of the batch.

To increase the amount of parallelism when processing batches of transactions, we parallelise the search for graph patterns for input batch transactions using a fine-grained approach, as illustrated in Figure 5.3. Because the search for graph patterns is usually performed using a recursive function (see Section 1.2), our library explores a recursion tree for each edge e that represents a transaction from the input batch. As an example, our library can detect

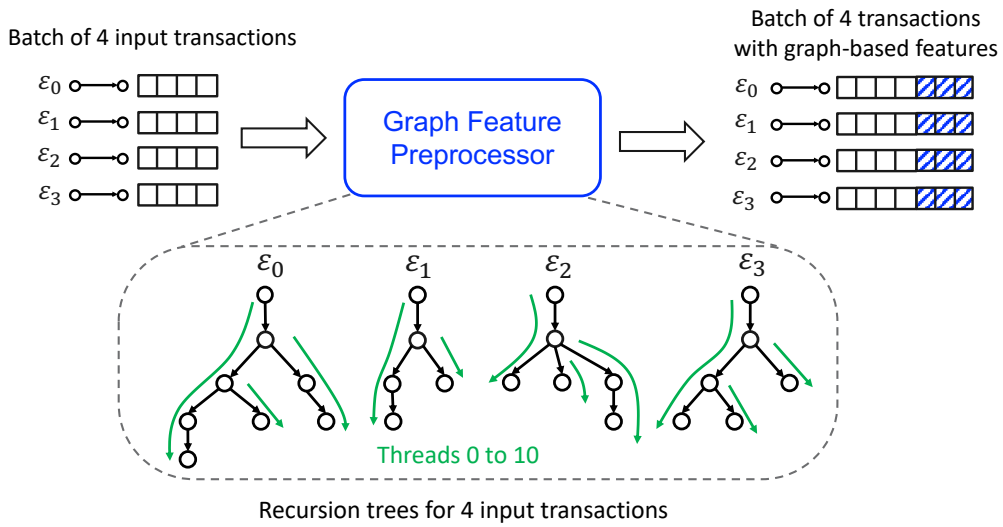


Figure 5.3: Fine-grained parallelism exploited by our *Graph Feature Preprocessor* library. The library searches for graph patterns independently for each input transaction by recursively exploring the transaction graph. The coarse-grained approach would use only four threads, while the fine-grained approach uses eleven threads.

whether an edge e belongs to a cycle by performing a recursive search starting from this edge e, as explained in Chapter 4. Our fine-grained algorithms presented in that chapter are able to execute the same recursion tree using several threads, thereby increasing the parallelism. Parallelising the search for graph patterns using a fine-grained approach enables us to process the recursion trees shown in Figure 5.3 with more threads compared to the case that uses a coarse-grained parallelisation approach. As a result, even if the input batch contains one transaction, our library would be able to parallelise the search for graph patterns.

5.2 Feature Extraction Library

An overview of our library is given in Figure 5.4. This library operates in a streaming fashion, receiving as input a batch of transactions with only basic features such as in Figure 5.1, and producing additional graph-based features as output. The library stores past financial transactions in an in-memory directed multigraph which is dynamically updated as new transactions are received (see Section 5.3 for further details). The graph-based features are computed by enumerating various subgraph patterns in the in-memory multigraph and by generating statistical properties of the accounts stored in that multigraph. The library can compute the graph-based features across several CPU cores in parallel, which, together with the dynamic multigraph representation, enables real-time feature extraction, as demonstrated in Section 5.5.

We have implemented our graph feature extraction library as a scikit-learn-compatible *fit*/*transform* interface used for preprocessing data [dev22a; dev22b] and made it publicly avail-

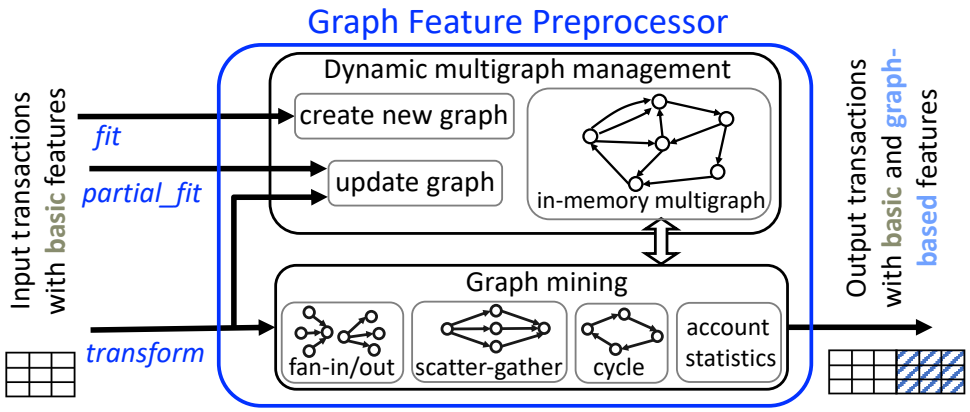


Figure 5.4: Block diagram of our graph feature extraction library. It supports scikit-learn-compatible *fit*/*transform* interface, which enables it to be integrated into existing scikit-learn pipelines for model training and scoring.

able on PyPI as part of the Snap ML package [IBM22b; IBM22c; IBM22a]. The main functionality of our library is implemented by the *transform* function, which is illustrated in Figure 5.4. This function inserts a batch of input transactions into the in-memory multigraph and computes graph-based features for these transactions (see below). Creating the initial in-memory multigraph is performed by providing some past transactions as an input to the *fit* function. The existing in-memory multigraph can be updated without computing any graph features by using the *partial_fit* function. Other standard preprocessor functions supported by our library are described in the publicly-available documentation [IBM22b].

Our library computes two main types of graph-based features: *(i)* graph-pattern-based features and *(ii)* account-statistics-based features. These features are produced when the *transform* function is called after setting the preprocessor parameters appropriately.

Graph-pattern-based features are computed for each transaction or account of the input batch by enumerating a predefined set of subgraph patterns that contain this transaction or account in the financial transaction graph. The patterns currently supported are fan-in, fan-out, gather-scatter and scatter-gather patterns as well as simple and temporal cycles (see Figure 5.5 for examples).

A *fan-out* pattern of vertex v is a subgraph pattern defined by the outgoing edges of v that connect v to $k \geq 2$ different vertices [SK21]. Analogously, in a *fan-in* pattern of vertex v is connected to $k \geq 2$ different vertices through the incoming edges of v. Examples of a fan-in and a fan-out patterns are given in Figures 5.5a and 5.5b, respectively. A *gather-scatter* pattern combines a fan-in pattern of the vertex v with a fan-out pattern of the same vertex v, as illustrated in Figure 5.5c [Sta+21]. Our library implicitly enumerates gather-scatter patterns by finding fan-in and fan-out patterns of the same vertex. There exists a vertex in each of the aforementioned subgraph patterns that can reach every other vertex of the same pattern in a single hop. Thus, we refer to these patterns as single-hop patterns.

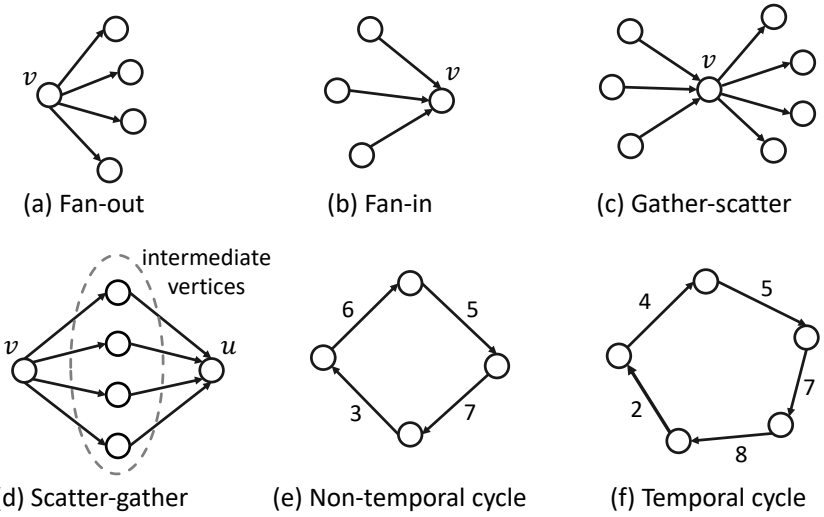


Figure 5.5: Examples of subgraph patterns that can be enumerated by our graph feature extraction library. The single-hop patterns supported are shown in (a), (b), and (c), and the multi-hop patterns supported are shown in (d), (e), and (f).

Currently, the multi-hop patterns that can be enumerated by our library are limited to scatter-gather patterns, simple cycles, and temporal cycles. A fan-out pattern of a vertex v and a fan-in pattern of a vertex u form a *scatter-gather* pattern if the fan-out and the fan-in patterns connect vertices v and u, respectively, to the same set of intermediate vertices [Sta+21]. Figure 5.5d shows an example of a scatter-gather pattern with four intermediate vertices. Examples of non-temporal and temporal cycles are given in Figures 5.5e and 5.5f, respectively.

To reduce the complexity, our library enumerates patterns in shifting time windows, the size of which can be specified by users. When the time-window size is δ for a pattern, the *transform* function enumerates patterns that consist only of edges with timestamps in $[t_{now} - \delta : t_{now}]$, where t_{now} is the minimum timestamp the *transform* function receives in the current batch of transactions.

Account-statistics-based features are computed only for the accounts that appear in the input batch of transactions. Note that each such account corresponds to a vertex of the dynamic in-memory multigraph. For each such account, some pre-defined statistical features can be computed for both its outgoing edges and its incoming edges. A selection of such features can be enabled or disabled by setting the preprocessor parameters appropriately. The statistical features currently supported by our library are: sum, mean, minimum, maximum, median, variance, skew, and kurtosis [KZ00]. These additional features can be computed and reported independently for each basic transaction feature. For instance, computation of the *sum* feature for the account B shown in Figure 5.1 using the basic feature "Amount" of the transactions that contain B as the source account, results in a feature value of 500. Combining different statistical feature types with different user-specified basic features in this way extends the feature space significantly.

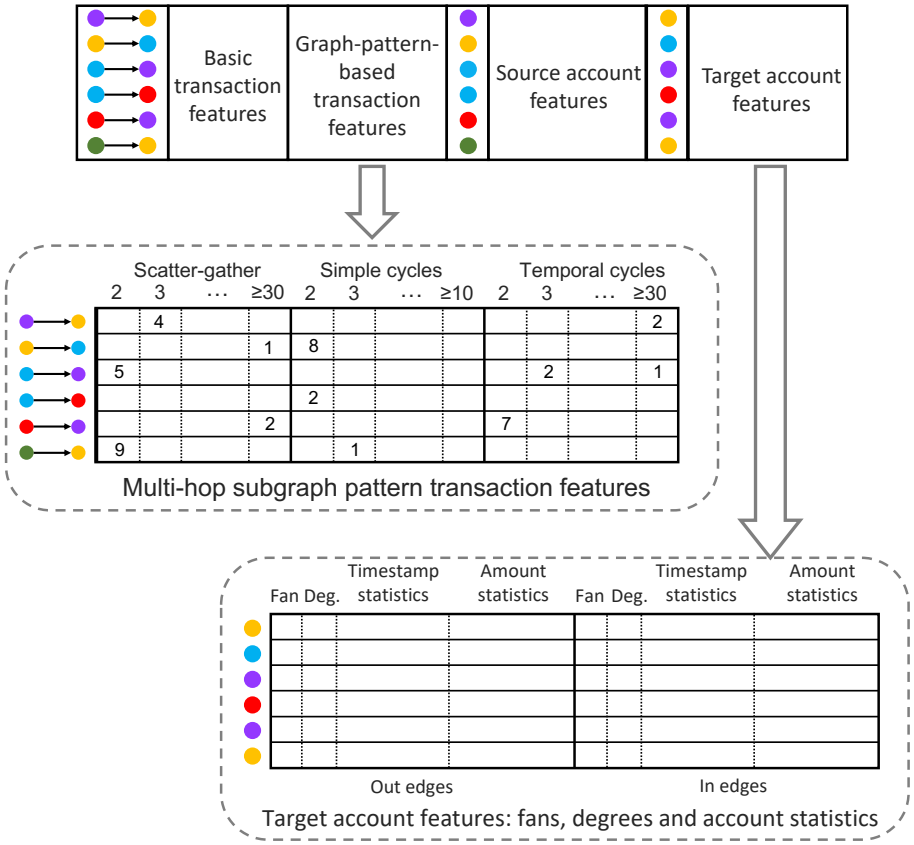


Figure 5.6: Feature encoding: scatter-gather patterns are binned according to the number of intermediate vertices they have and cycles are binned according to their length. Basic features used for the computation of account-statistics-based features are "Timestamp" and "Amount".

The encoding of the features produced by the *transform* function is shown in Figure 5.6. As with the input feature table in Figure 5.1, each row of the output feature table stores the feature vector of a single transaction. Across different columns of a feature vector, there are basic transaction features, graph-pattern-based transaction features, and the account features of the source and the destination account of the transaction. The account features consist of account-statistics-based features and features based on fan-in and fan-out patterns, both of which are single-hop patterns. Features based on fan-in and fan-out patterns are computed for each account v and represent the number of accounts connected to v in those patterns. Graph-pattern-based transaction features are computed using multi-hop subgraph patterns. For each transaction, our library reports the number of multi-hop subgraph patterns of different sizes that this transaction is part of. Example features based on multi-hop subgraph patterns are given in Figure 5.6, where the first transaction participates in 4 scatter-gather patterns with 3 intermediate vertices and in 2 temporal cycles with 30 or more edges. Even though the multi-hop subgraph patterns can also be used to compute account features, computing them as transaction features provides more compact feature vectors.

5.3 Dynamic Multigraph Support

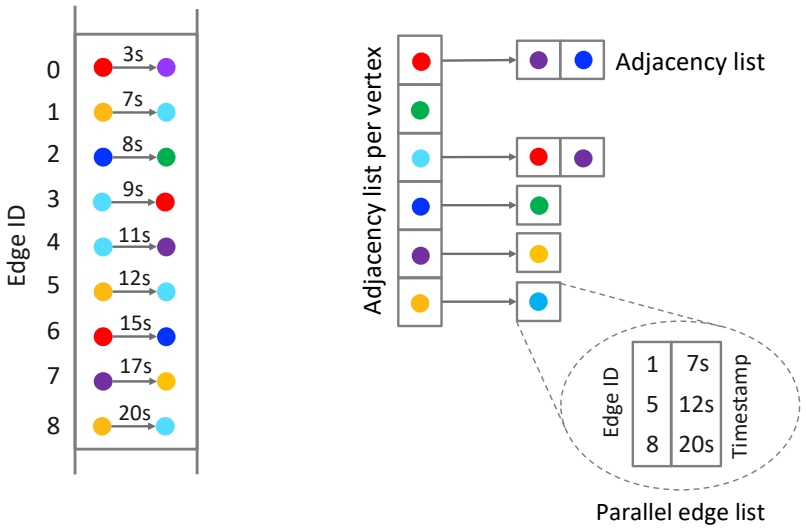
To support real-time feature extraction in financial transaction graphs, we developed a dynamic in-memory temporal multigraph representation. Our representation enables fast insertion of edges, fast identification and removal of the edge with the smallest timestamp, fast access to the neighbours of a vertex, and support for maintaining parallel edges, i.e., edges with the same source and destination vertices. Fast constant-time insertions are required to enable real-time processing of input batches of transactions, and fast accesses to the neighbours of a vertex are crucial for the performance of graph mining algorithms. Support for maintaining parallel edges is required given that there could exist several transactions between the two accounts in the financial transaction graphs.

To reduce the memory usage of our dynamic in-memory multigraph representation, the outdated edges with timestamps outside the time window $[t_{now} - \delta : t_{now}]$ are removed, where t_{now} is the timestamp of the last inserted edge, and δ is the user-specified time window size. To remove these edges, a fast operation that removes the edge with the smallest timestamp in the graph is required. The removal of edges outside of the time window $[t_{now} - \delta : t_{now}]$ also reduces the number of edges visited during the execution of the graph mining algorithms, resulting in their faster execution.

5.3.1 Data Structures

The overview of our dynamic multigraph representation is given in Figure 5.7. It consists of two main parts: a transaction log (Figure 5.7a) and an index (Figure 5.7b). The transaction log maintains a list of edges sorted in the ascending order of their timestamps. This data structure only stores edges that have timestamps in the time window $[t_{now} - \delta : t_{now}]$. Having a sorted list helps facilitate removal of the edges that fall outside of this time window. To implement the transaction log, we use a double-ended queue (i.e., *deque*), which enables constant-time operations that insert edges to or remove edges from the front or back of the deque [Knu68; ref23a]. Because the edges arrive ordered in time, the new edges are simply inserted into the back of the deque. In addition, the edge with the smallest timestamp in the transaction log can be removed from the front of the deque in constant time. As a result, the outdated edges can then simply be removed by repeatedly removing the edge with the smallest timestamp from the transaction log.

The index data structure uses an adjacency list representation to enable fast accesses to the neighbours of a vertex [Cor09a], as illustrated in Figure 5.7b. This data structure is implemented as a vector of hash maps [ref23b], where each entry in the vector represents a vertex v and the hash map associated with that vertex v represents the adjacency lists of v. The vertices are internally mapped to integers in the range of $0, 1, \ldots, n-1$, where n is the number of vertices in the graph, and these integers are used to access the adjacency list of a vertex v in this vector. Note that our dynamic in-memory multigraph maintains two different index data structures, one in which the adjacency lists represent outgoing vertices and one in which



(a) Transaction log (b) Index

Figure 5.7: Our in-memory dynamic multigraph. Transaction log from (a) maintains a sorted edge list and the index in (b) represents an adjacency list for each vertex in the graph. Parallel edges are represented as a list of edge IDs and their respective timestamps.

the adjacency list represents incoming vertices. Thus, accessing both outgoing and incoming neighbours of a vertex can be performed in constant time.

To support maintaining parallel edges in the index, each entry in an adjacency list of the vertex v that represents a neighbour u of the vertex v also contains a list of edges that connect v with u. We refer to this data structure as *parallel edge list* (see Figure 5.7b). The edges of this list are represented with their ID and timestamp and are sorted in ascending order of their timestamps. The parallel edge list is implemented as a deque, which enables inserting an edge to the back of the list and removing an edge from the front of the list, both in constant time. Given that edges are inserted into the graph in the increasing order of their timestamps and that the edges with the smallest timestamp are always removed first, the edges in parallel edge lists stay sorted after updating the list. Because accessing the parallel edge list of the vertex u in the adjacency list of a vertex v is performed in constant time, inserting and removing an edge that connects v with u is also performed in constant time. Therefore, both insert and remove operations of our dynamic multigraph are performed in $O(1)$ time.

Our feature extraction library supports saving and loading the state of the dynamic in-memory multigraph [doc23]. Because the index can be reconstructed from the transaction log, it is sufficient to save the edges stored in the transaction log when saving the state of the library. Upon loading the state, the saved edges are inserted into the transaction log and the index in ascending order of their timestamps, as explained above. Note that the transaction log can be mapped to a memory-mapped file to accelerate the process of saving and loading state, as well as to reduce memory usage of our dynamic in-memory multigraph. However, our feature extraction library does not yet support this option.

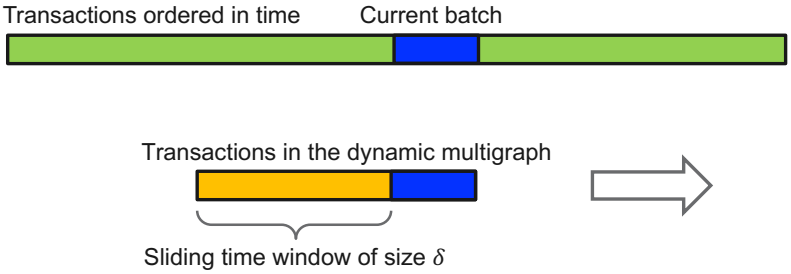


Figure 5.8: Stream processing: after a batch of transactions is inserted into the dynamic multigraph, our library extracts the graph-based features for those transactions and removes transactions that fall outside of the sliding time window.

Algorithm 15: ScatterGatherStream $(\mathcal{G}(\mathcal{V},\mathcal{E}), batch, \delta)$

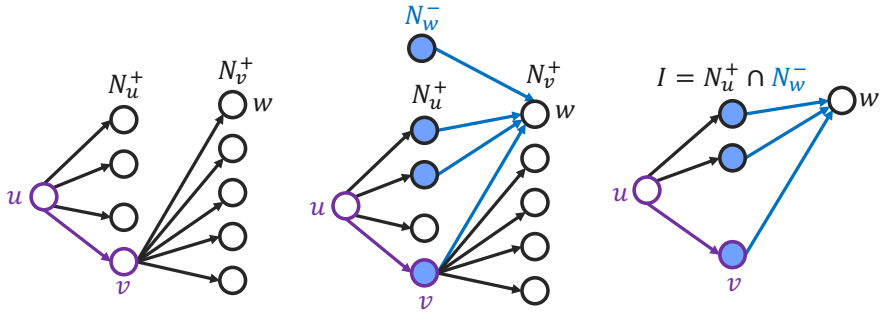
Input: $\mathcal{G}$ - the input graph with edges $\mathcal{V}$ and edges $\mathcal{E}$
$batch$ - vector containing the batch of input edges
δ - the time window

1 **parallel foreach** (u → v, t_{uv}) *: batch* **do**
2 TW = $[t_{uv} - \delta : t_{uv}]$; ▷ Time window of size δ
3 $N_u^+ = \{\forall x \mid (u \rightarrow x,\ t_s) \in \mathcal{E} \wedge t_s \in \text{TW}\}$; ▷ The first phase
4 $N_v^+ = \{\forall x \mid (v \rightarrow x,\ t_s) \in \mathcal{E} \wedge t_s \in \text{TW}\}$;
5 **parallel foreach** w : N_v^+ **do**
6 $N_w^- = \{\forall x \mid (x \rightarrow w,\ t_s) \in \mathcal{E} \wedge t_s \in \text{TW}\}$;
7 $I = N_u^+ \cap N_w^-$;
8 **if** $|I| \geq 2$ **then** report scatter-gather pattern $\{u, I, w\}$;
9 $N_u^- = \{\forall x \mid (x \rightarrow u,\ t_s) \in \mathcal{E} \wedge t_s \in \text{TW}\}$; ▷ The second phase
10 $N_v^- = \{\forall x \mid (x \rightarrow v,\ t_s) \in \mathcal{E} \wedge t_s \in \text{TW}\}$;
11 **parallel foreach** w : N_u^- **do**
12 $N_w^+ = \{\forall x \mid (w \rightarrow x,\ t_s) \in \mathcal{E} \wedge t_s \in \text{TW}\}$;
13 $I = N_v^- \cap N_w^+$;
14 **if** $|I| \geq 2$ **then** report scatter-gather pattern $\{w, I, v\}$;

5.3.2 Stream Processing

This dynamic multigraph representation enables our feature extraction library to operate in a streaming manner, as illustrated in Figure 5.8. In this setting, our library processes transactions in batches by inserting each batch into the dynamic graph and extracting the graph-based features for transactions in that batch using our dynamic in-memory multigraph. After processing a batch, the time window $[t_{now} - \delta : t_{now}]$ is updated by setting the value t_{now} to the largest timestamp among the transactions in that batch. As a result, the transactions of the dynamic multigraph that have timestamps smaller than $t_{now} - \delta$ are removed.

To compute graph-pattern-based features in a streaming manner, our library enumerates new patterns that are formed after inserting the input batch of edges into the graph. The fan-out pattern feature of a vertex v that belongs to the input batch is determined by counting the



(a) Determine N_u^+ and N_v^+ (b) Determine N_w^- (c) The resulting pattern

Figure 5.9: Enumeraton of scatter-gather patterns that contain the edge $u \rightarrow v$ with v being an intermediate vertex. Similarly to other algorithms for graph pattern mining, this approach also uses set intersections to determine vertices that belong to a subgraph pattern.

number of the outgoing vertices of v. Similarly, the fan-in pattern feature of v is computed by counting the number of the incoming vertices of v. Both fan-out and fan-in features can be determined in $O(1)$ time by simply querying the size of the hash maps that are implementing the adjacency lists of the vertex v in our index data structure (see Section 5.3.1).

To enumerate simple and temporal cycles, we use fine-grained parallel algorithms introduced in Chapter 4 of this thesis. These algorithms enable the search for cycles that start from a single edge or a small batch of edges in parallel using several threads. This approach is illustrated in Figure 5.3.

To compute scatter-gather pattern in a streaming manner, we use our algorithm illustrated in Figure 5.9 and presented in Algorithm 15. This algorithm processes each edge $(u \rightarrow v, t_{uv})$ in the input batch by searching for all scatter-gather patterns that include the edge the $(u \rightarrow v, t_{uv})$ and consist of edges that have a timestamp within the time window $TW = [t_{uv} - \delta : t_{uv}]$, where δ is a user-defined time window size. There are two phases of this algorithm: the first phase that searches for scatter-gather patterns with v as an intermediate vertex, and the second phase that searches for scatter-gather patterns with u as an intermediate vertex. In the first phase, the algorithm first determines the outgoing neighbours of u and v, denoted as N_u^+ and N_v^+, respectively, as shown in Figure 5.9a. Then, for each neighbour w of v, the algorithm searches for incoming neighbours N_w^- of the vertex w, which are represented as filled circles in Figure 5.9b. Only the edges that have timestamps within the time window TW are considered when determining sets N_u^+, N_v^+, and N_w^-. Afterwards, the algorithm performs a set intersection between N_u^+ and N_w^-, and the resulting vertices represent the intermediate vertices I of a scatter gather pattern. Finally, the algorithm reports the resulting scatter-gather pattern defined with vertices u, w, and I, as shown in Figure 5.9c. The second phase of this algorithm, presented in lines 9–14 of Algorithm 15, is analogous to the first phase, and we omit its description for brevity.

Our algorithm for enumerating scatter-gather patterns executes in $O\left(|batch| \times \Delta_{max}^2\right)$ time, where Δ_{max} is the maximum degree of a vertex in the graph and $|batch|$ is the number of edges

in the input batch. This time complexity bound can be derived by observing that the vertex sets produced in the algorithm contain at most Δ_{max} vertices and that the set intersection between these sets can be performed in $O(\Delta_{max})$. This algorithm can be accelerated by parallelising its loops, as shown in Algorithm 15. The iterations of inner loops, shown in lines 5 and 11 of Algorithm 15, are independent of each other and can be executed concurrently. Thus, we can exploit fine-grained parallelism by processing a single edge from the input batch using several threads. As a result, the depth of this algorithm, which is the time needed to execute this algorithm using an infinite number of threads, is $T_\infty = O(\Delta_{max})$.

Stream processing enables our library to incrementally compute account-statistics-based features by updating the statistical properties of a vertex after an edge is inserted or removed. For this purpose, our library maintains second, third, and fourth central moments for each vertex of the graph and for each basic feature used for calculating account statistics (e.g., "Amount"). After inserting or removing an edge $u \rightarrow v$, all central moments for u and v are updated incrementally [Fin09; TK12]. These central moments are then used to compute the following statistical features: sum, mean, variance, skew, and kurtosis [KZ00]. Note that the computation of each aforementioned statistical feature can be computed in $O(1)$ time. Other statistical features, i.e., minimum, maximum, and median, are simply computed by iterating through the incident edges of a vertex, which is executed in $O(\Delta_{max})$ time per each statistical feature.

5.4 Graph Machine Learning Pipeline

This section describes how to train a machine learning model used in our graph machine learning pipeline shown in Figure 5.2. In addition, we explain how the inference is performed using the trained model and our *Graph Feature Preprocessor* library.

The training step of our graph machine learning pipeline is illustrated in Figure 5.10. First, the transactions available for training are ordered in ascending order of their timestamps and are split into train, validation, and test sets. This split is performed in such a way that the transactions from the train set have the lowest timestamps and the transactions from the test set have the highest. Then, the transactions from the train and validation sets are forwarded to the *Graph Feature Preprocessor* to generate the enriched graph-based features for the transactions from these two sets. To prevent any form of information leakage at training time, the training set is processed before the validation set. In that case, graph-based features for the transactions of the train set are computed on the graph created using only those transactions, and thus no information from the validation set is used. Finally, the train and validation sets with enriched features are then used to train the gradient boosting models [Ke+17; CG16]. The test set is not processed at this step and is used for the pipeline evaluation.

Training a machine learning model often involves hyper-parameter tuning, which requires both the train and validation sets. The train set is used to train ML models with different hyper-parameters, which are evaluated on the validation set. The hyper-parameters that give

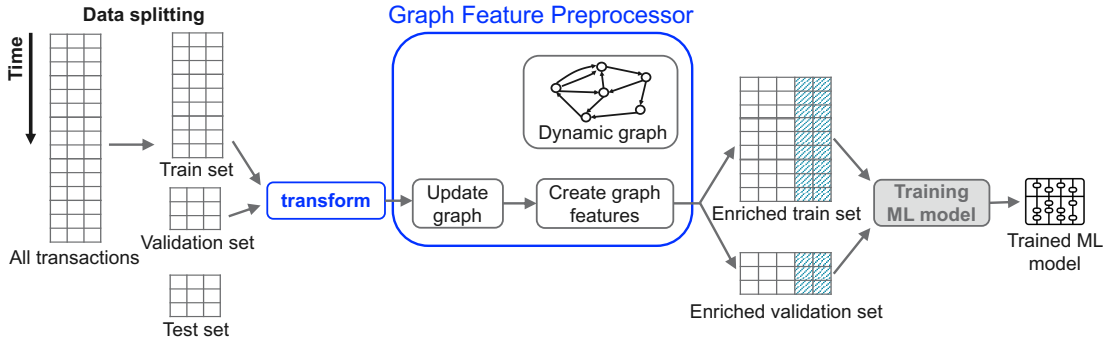


Figure 5.10: The training step of our graph machine learning pipeline shown in Figure 5.2. The model used in our pipeline is trained using graph-based features produced by our *Graph Feature Preprocessor* library.

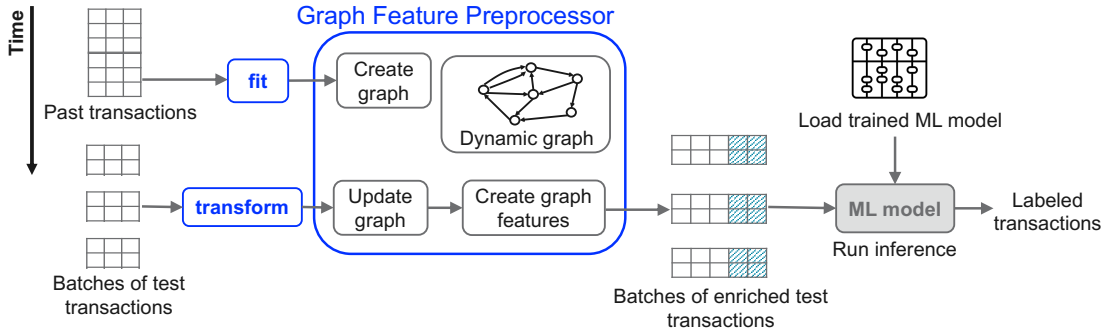


Figure 5.11: The inference step of our graph machine learning pipeline shown in Figure 5.2. The transactions are processed in small batches. The transactions with graph-based features are forwarded to a model trained using the setup from Figure 5.10, which labels transactions as licit or illicit.

the best accuracy on the validation set are chosen to train a model using both the train and the validation sets. The resulting model is then used for inference in our graph machine learning pipeline.

The inference step of our graph machine learning pipeline is shown in Figure 5.11. First, we load the model trained using the setup shown in Figure 5.10. Then, we initialise the *Graph Feature Preprocessor* library by loading past financial transactions using the *fit* function. These past financial transactions are used to create the initial dynamic graph. Next, the transactions from the test set are grouped into batches and forwarded to the *Graph Feature Preprocessor* using the *transform* function. This function updates the existing dynamic graph using the forwarded transactions and enriches those transactions with graph-based features of the same type as those generated in the train setup (see Figure 5.10). Finally, the enriched test transactions are sent to the pre-trained machine learning model for detection of transactions associated with financial crime.

Table 5.1: Datasets used in the experiments. Illicit rate refers to the percentage of illicit transactions and time span refers to the difference between the maximum and minimum timestamps in a dataset.

Dataset	No. nodes	No. edges	Illicit rate	Time span [days]
AML 100M	1.8 M	100 M	0.17%	180
AML HI Small	0.5 M	5 M	0.07%	10
AML HI Medium	2.1 M	32 M	0.11%	16
AML HI Large	2.1 M	180 M	0.11%	97
ETH Phishing	2.9 M	13 M	0.27%	1261

5.5 Experimental Evaluation

In this section, we evaluate the performance of our feature extraction library as well as the accuracy of our graph machine learning pipeline.

5.5.1 Experimental Setup

Datasets. The datasets used in the evaluation are presented in Table 5.1. The AML HI datasets are the publicly available synthetic AML datasets available on Kaggle [IBM23c] whereas the AML 100M dataset is a proprietary synthetic AML dataset. The AML datasets contain transactions labelled as laundering or not laundering, and, thus these datasets can be directly used with our Graph ML pipeline that performs transaction classification. The ETH Phishing dataset is a real-world Ethereum dataset [Che+19a] which contains 1,165 accounts labelled phishing. To enable transaction classification using the ETH Phishing dataset, we label a transaction of this dataset as phishing if at least one of its destination accounts is labelled as phishing. As a result, 36,055 transactions out of 13 M are labelled as phishing.

ML model parameter tuning. We use the aforementioned datasets to train LightGBM [Ke+17] and XGBoost [CG16] boosting machines, which are widely used ML models for tabular data. These models are defined by a large number of parameters that need to be carefully tuned, especially when dealing with highly imbalanced datasets as is typically the case of AML financial datasets. Evaluating many parameter combinations on large datasets can be time- and resource-expensive. To address this challenge, here we employ a successive halving (SH) model tuning approach [JN14] where the SH resource is defined as the number of examples. SH is a multi-round algorithm that starts by randomly sampling x_0 model parameter combinations which are evaluated using a small percentage of the train examples, e.g., $r_0 = 0.1$ (10% of the training examples). The algorithm then finds the best $\eta^{-1} \times x_0$ configurations ($\eta > 1$). These are used in the next SH round where each of the $\eta^{-1} \times x_0$ configurations is trained on $\eta \times r_0$ train examples. SH continues by decreasing the number of parameter configurations and by increasing the number of train examples until the maximum number of train examples has been reached. In our experiments, we used different SH configurations for different data sets: $x_0 = 16$, $\eta = 2$, $r_0 = 0.2$ for the large AML data sets, and $x_0 = 1000$, $\eta = 2$, $r_0 = 0.1$, and $x_0 = 100$, $\eta = 2$, $r_0 = 0.2$ for the smaller AML and ETH Phishing data sets. Details about the

Table 5.2: Model parameter ranges used at tuning time.

LightGBM		**XGBoost**	
Parameter	**Range**	**Parameter**	**Range**
num_round	(10, 1000)	num_round	(10, 1000)
num_leaves	(1, 16384)	max_depth	(1, 15)
learning_rate	$10^{(-2.5,-1)}$	learning_rate	$10^{(-2.5,-1)}$
lambda_l2	$10^{(-2,2)}$	lambda	$10^{(-2,2)}$
scale_pos_weight	(1, 10)	scale_pos_weight	(1, 10)
lambda_l1	$10^{(0.01,0.5)}$	colsample_bytree	(0.5, 1.0)
		subsample	(0.5, 1.0)
early_stopping_rounds = 20			

model parameter ranges are shown in Table 5.2.

Train/Validation/Test split. To tune the parameters of the models and to test the model generalization performance, we split the input data into train, validation, and test sets. The train and validation sets are used by the SH scheme to tune the model, while the test set is used for the final evaluation of the model. The splitting is performed in a temporal manner using two timestamps T_1 and T_2, $T_1 < T_2$, such that the train set contains the transaction with timestamps smaller than T_1, the validation set contains the transactions with timestamps with values between T_1 and T_2, and the test set contains the rest of the transactions. For the AML datasets, we determine T_1 and T_2 such that the train, validation, and test sets contain approximately 60%, 20%, and 20% of the data, respectively, and that any two transactions that were created on the same day are placed in the same set. For the ETH Phishing dataset, we define the timestamp of an account as the minimum timestamp among the transactions that involve this account and determine T_1 and T_2 such that 65% of the accounts have timestamps smaller than T_1 and 15% of the accounts have timestamps with values between T_1 and T_2. The transactions in the train, validation, and test sets are then determined using T_1 and T_2 as indicated above.

Data leakage at train/tune time. We extract different sets of graph-based features from each dataset using our feature extraction library. For each dataset, we sort the transactions in the increasing order of their timestamp and send batches of transactions to the graph feature extraction library. Data leakage is prevented by this ordering because only the past data is used during feature extraction. Furthermore, for the ETH Phishing dataset, we temporally split the accounts rather than the transactions to prevent data leakage that could occur if the same phishing account is involved in the transactions from different sets.

Graph ML pipeline as an ML service setup. We evaluate the performance of the graph machine learning inference pipeline deployed as an ML service using a client-server setup implemented on an IBM[I] z16 system [IBM23b] that uses the IBM Telum processor [Lic+22]. The server is a logical partition with Ubuntu 20.04, 6 dedicated IBM Integrated Facility for Linux processors

[I]IBM, the IBM logo, IBM Cloud Pak, and IBM Telum are trademarks or registered trademarks of International Business Machines Corporation, in the United States and/or other countries.

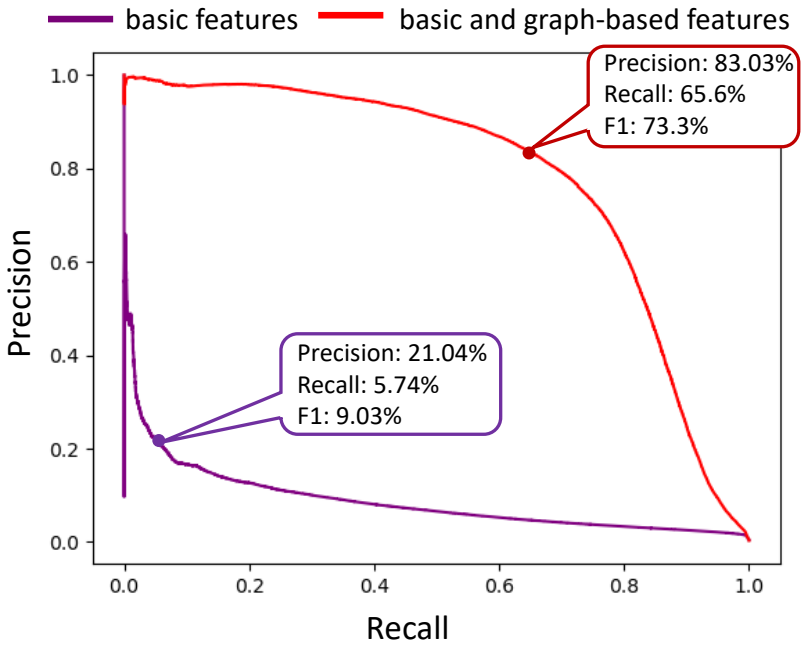


Figure 5.12: Precision-recall curves for LightGBM models performing money laundering detection using AML 100M. For the prediction threshold of 0.5, our graph-based features increase precision by 62% and recall by 60%, resulting in a 64% increase in the F1 score for this problem.

Table 5.3: Performance of the client-server z16 setup using the AML 100M dataset.

	Throughput [tps]	**Latency [ms]**
Feature extraction	124,272	1.03 ms
Inference	185,507	0.69 ms
End-to-end	34,133	3.75 ms

(IFL) and 256 GB memory. The client is a separate logical partition with Ubuntu 21.04, 12 dedicated IFLs and 64 GB memory. The client and server images communicate via an internal high-speed network. This setup is used in the experiments described in Section 5.5.2.

Setup for the evaluation on public datasets. We evaluate our graph ML pipeline, shown in Figure 5.2, using the publicly available AML HI and Ethereum datasets introduced in Table 5.1. The performance of our pipeline is evaluated using 16 cores of an Intel Xeon E5-2667 v2 processor. To evaluate performance, we first invoke the *fit* function (see Section 5.2) using the transactions from the train and validation sets, which creates the initial graph. Then, we invoke the *transform* function of the library using batches of transactions from the test set, which are forwarded to a pre-trained ML model (see Figure 5.11). We measure the average latency of performing the *transform* function together with the prediction of the ML model. Finally, we compute the throughput of our pipeline as the transactions processed per second (tps). This setup is used in the experiments described in Section 5.5.3.

5.5.2 Graph ML Pipeline as a z16 Service

To detect fraudulent financial transactions in a production system, the Graph ML inference pipeline described in Section 5.4 would be deployed as a service using an ML serving framework, such as BentoML, KServe etc. A user of the service would send single or batches of financial transactions in the form of, e.g., HTTP requests, to the inference service, which would run the inference pipeline and return the predictions back to the user.

To test such a production-like scenario, we used the client-server z16 setup described in Section 5.5.1. The client is a Python-based custom workload generator that simulates a user that sequentially reads batches of transactions from memory and sends them to the server for prediction using the REST API. The server runs BentoML [Ben22] as an ML serving framework which exposes the Graph ML inference pipeline as a service. For each user request, the BentoML serving logic implements two steps: graph feature extraction and transaction classification. The latter is performed using a tuned LightGBM model. The client measures the end-to-end latency of querying the ML service, which includes not only the feature extraction and ML model classification latency, but also the network and the overheads associated with the ML serving framework.

We evaluated the z16 client-server setup using the proprietary AML 100M dataset. When extracting graph-based features from this dataset, we used a batch size of 128 and a time window of 20 days. Scatter-gather patterns are searched in a five-day time window, simple cycles in a 10-day time-window, and temporal cycles in a 20-day time window. We used the "Amount" and "Timestamp" fields of the basic transaction features to generate the account statistics.

Figure 5.12 compares the precision-recall curve of the LightGBM model that uses only basic transaction features with that of the model that uses our graph-based features in addition to the basic features. We observe that the features computed by our graph feature extraction library improve the minority-class F1 score of LightGBM from 9% to 73%. The latency and the throughput values we have measured are given in Table 5.3. Our library processes a batch of 128 transactions in around 1 ms, enabling our library to achieve a throughput rate exceeding 100,000 tps. The inference latency of the LightGBM model that uses our graph-based feature is 0.69 ms. The end-to-end latency of our client-server z16 setup, which also includes network overheads, is 3.75 ms. Such a low end-to-end latency shows that our graph feature extraction library can help detect laundering transactions in a real-time setting.

5.5.3 Experiments on Public Datasets

In this section, we evaluate the accuracy of the LightGBM and XGBoost models trained on the publicly available datasets from Table 5.1 with and without our graph-based features. We also report the performance of our graph-based feature extraction library when using the publicly available AML HI datasets. The features are extracted from the AML HI datasets using

Table 5.4: Minority class F1 scores of 1) the money laundering detection task using the AML datasets and 2) of the phishing detection task using the ETH Phishing dataset, where bf and gf stand for basic features and graph-based features, respectively. BS stands for batch size. NA stands for not available.

	BS	AML HI Small	AML HI Medium	AML HI Large	**BS**	ETH Phishing
LightGBM bf	—	21.3±0.3%	18.6±0.1%	24.5±0.2%	—	13.7±0.5%
LightGBM bf+gf	128	62.1±0.4%	54.8±0.5%	48.7±0.2%	128	40.2±0.2%
LightGBM bf+gf	2048	60.5±0.6%	56.1±0.4%	46.6±0.3%	∞	51.0±1.0%
XGBoost bf	—	19.7±0.8%	20.1±0.2%	10.6±6.7%	—	15.5±0.1%
XGBoost bf+gf	128	63.9±0.2%	60.5±0.2%	50.2±0.6%	128	37.0±2.0%
XGBoost bf+gf	2048	64.7±0.4%	59.1±0.2%	56.8±0.2%	∞	49.4±0.5%
GIN	∞	24.4±4.6%	40.1±2.7%	NA	∞	35.1±4.1%
LaundroGraph	∞	40.7±5.8%	56.0±1.5%	NA	∞	40.2±4.2%
PNA	∞	51.9±4.6%	68.1±2.6%	NA	∞	51.5±4.3%

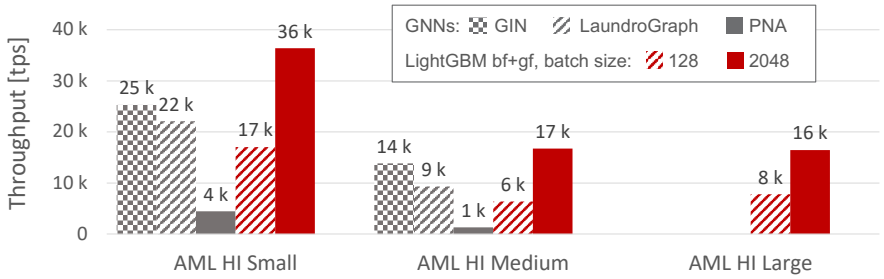


Figure 5.13: Throughput of our graph ML pipeline and the GNN baselines, where bf and gf stand for basic features and graph-based features, respectively. Our graph ML pipeline uses LightGBM for inference. Our solution has higher throughput than the GNN-based approaches.

two different batch sizes: 128 and 2048. We use a time window of six hours for scatter-gather patterns and a time window of one day for the rest of the graph-based features. We specify a hop constraint of 10 for simple cycle enumeration. We use the "Amount" and "Timestamp" fields of the basic transaction features to generate the account statistics. Feature extraction from the ETH Phishing dataset is performed using two batch sizes: 128 and ∞. In addition, a 20 day time window is used. When using a batch size of ∞, all transactions of the test set are made available to the graph feature extraction library in a single batch. In addition, we disable the generation of temporal cycles and specify a hop constraint of 5 for simple cycle enumeration. We use the "Amount", "Timestamp", and "Block Nr." fields of the basic transaction features to generate the account statistics. We selected these parameters after some careful exploration aimed at finding the best trade-offs between the throughput of the feature extraction library and the accuracy of the ML models used for scoring.

Using a batch size of ∞ essentially corresponds to an offline solution and, in principle, can lead to better accuracy because, in this case, future transactions are also visible during feature extraction. However, if a real-time processing capability is required by an application, the batch size will have to be constrained.

Table 5.5: Latency of processing a single batch of transactions. Our graph ML pipeline uses LightGBM for inference and both basic features (bf) and graph-based features (gf). NA stands for not available. Our graph ML pipeline that uses batch size of 128 has the lowest per-batch latency.

	GIN	**LaundroGraph**	**PNA**	**LightGBM bf+gf**	
batch size	∞	∞	∞	128	2048
AML HI Small	33 s	38 s	189 s	7 ms	56 ms
AML HI Medium	488 s	726 s	5058 s	20 ms	122 ms
AML HI Large	NA	NA	NA	16 ms	129 ms

Our baselines are the LightGBM and XGBoost models, as well as graph-neural-networks (GNNs) such as GIN [Xu+18], LaundroGraph [CSB22], and PNA [Cor+20]. These baselines are trained using only basic transaction features. Note that LaundroGraph is specifically designed for anti-money laundering, and uses additional neural network layers that perform edge updates in addition to node updates to derive better transaction embeddings. However, GIN, LaundroGraph, and PNA require the entire dataset to be available at the time of testing, and, thus, cannot operate in a streaming manner and are not suitable for real-time processing. Effectively, they use a batch size of ∞.

Furthermore, when training our machine learning models, we remove the source and destination account IDs from the feature vectors to prevent the models from learning which transactions are laundering/phishing by simply memorising the account IDs. As our measure of accuracy, we use the minority-class F1 score. The F1 scores reported are averaged across five different runs. The standard deviation of the F1 score is also reported for each run.

The minority class F1 scores of the ML models that perform laundering detection using publicly available AML HI datasets are shown in Table 5.4. Clearly, our graph-based features lead to significant improvements in the F1 scores achieved by gradient boosting models. Without our graph-based features, the maximum F1 score achieved by LightGBM and XGBoost is 24.5%. The reason is that the labels in all datasets are highly imbalanced, and the number of illicit transactions is at most 0.11% of the total number of transactions (see Table 5.1). The LightGBM and XGBoost models that use our graph-based features in addition to basic features achieve up to 43% higher F1 scores than the models that use only basic features. Furthermore, given that the F1 scores of our graph ML pipeline using either LightGBM or XGBoost do not change significantly when changing the batch size (see Table 5.4), choosing the appropriate batch size depends on whether higher throughput or lower latency is preferred. As we can see in Figure 5.13 and Table 5.5, the use of a larger batch size results in higher throughput of our graph ML pipeline at the cost of increased latency of processing a single batch. Overall, the latency of processing a batch of 128 transactions in AML HI datasets ranges from 7 ms to 20 ms, which enables our graph ML pipeline to be used in a setting that requires real-time processing of transactions.

Compared with GNN baselines, our graph ML pipeline that uses LightGBM and XGBoost achieves higher accuracy for AML HI Small and competitive accuracy for AML HI Medium. As

shown in Table 5.4, our pipeline that uses XGBoost for inference and our graph-based features consistently achieves higher F1 scores than GIN and LaundroGraph. Compared to PNA, our graph ML pipeline achieves up to a 12% higher F1 score for AML HI Small but lower F1 scores for AML HI Medium. However, as we can see in Figure 5.13, our graph ML pipeline has 8× and 12× higher throughput than PNA for AML HI Small and AML HI Medium, respectively. Furthermore, our graph ML pipeline is able to process transactions in a streaming manner with low latency, as shown in Table 5.5, which is not the case for the GNN baselines. Note that we were unable to obtain results for GIN, LaundroGraph, and PNA on the AML HI Large dataset due to the excessively long training runtimes and high memory requirements of these GNN-based approaches. Overall, our graph ML pipeline provides competitive accuracy with the GNN baselines while having higher throughput and the ability to operate in a streaming manner with low per-batch latency.

Table 5.4 also shows the minority class F1 scores achieved by the ML models we trained on the ETH Phishing dataset to perform phishing detection. When using a batch size of 128, our graph-based features enable F1-score improvements exceeding 20% for both LightGBM and XGBoost. Setting the batch size to ∞ further improves the F1 score of LightGBM to 51%. In that case, LightGBM with our graph-based features outperforms LaundroGraph by 10% and GIN by 16%. In addition, LightGBM with our graph-based features generated using batch size ∞ has accuracy competitive with that of PNA while achieving 20× higher throughput than PNA. Note that increasing the batch size from 128 to ∞ increases the per-batch processing latency of our graph ML pipeline from 59 ms to 345 s, effectively making it an offline solution. In general, the optimal configuration of the feature extraction library depends on the requirements of the end application, and might require trading off the performance for accuracy.

5.6 Related Work

Graph machine learning has applications in many different fields, including financial transaction network analysis [NKL21; Liu+21a; CS20; Wan+21], fraud detection [Liu+21b; Zhu+20b; Cao+19; Edd+22; Ama23; Cse+22], drug discovery [Gau+21], molecular property prediction [Zha+21], genomics [Sch+21], recommender systems [Eks+18], social network analysis [BGL16; Fan+19], and relation prediction in knowledge graphs [Qin+21]. Fraud detection systems TitAnt [Cao+19] and Eddin et al. [Edd+22] are graph machine learning systems that extract features from transaction graphs by generating node embeddings [PAS14] or by performing random walks [Oli+21] in transaction graphs. These features are then used by machine learning models to predict whether an incoming transaction is fraudulent or not.

Graph neural networks (GNNs) [Xu+18; Vel+18; Bou+23; KW17; HYL17; CSB22; Liu+21b; Wan+21; Bar+21] are powerful tools that can be used for the purpose of financial crime detection. Cardoso et al. [CSB22] and Weber et al. [Web+19] apply GNN to the anti-money laundering problem, Kanezashi et al. [Kan+22] apply GNN to the phishing detection problem on the Ethereum blockchain, and Rao et al. [Rao+21] uses a GNN to detect fraudulent transac-

tions. Graph Substructure Network, proposed by Bouritsas et al. [Bou+23], takes advantage of pre-calculated subgraph pattern counts to improve the expressivity of GNNs. GNNs could also be used to count subgraph patterns, such as in Chen et al. [Che+20b], which could enable detecting patterns associated with financial crime. In contrast to our work, GNNs cannot straightforwardly operate in a streaming manner and require the entire dataset to be available at the time of testing.

Dynamic graph management is often required for real-time processing of financial transactions. Dynamic graph data structures, such as STINGER [Edi+12], GraphTinker [JS19], and Sortledton [FMG22] enable dynamic insertions of edges into the graph as well as their removal from the graph. However, STINGER and GraphTinker cannot be directly used for representing financial transaction graphs because they do not support the maintenance of parallel edges. Furthermore, utilising hash maps to represent the adjacency list enables us to insert an edge into our dynamic in-memory multigraph in $O(1)$ time, whereas Sortedlon requires $O(\log(\Delta_{max}))$ time to perform edge insertion, with Δ_{max} being the maximum degree of a vertex in the graph. In-memory graph databases [Zhu+20a; Bur+20; Car+19] can also be used for dynamic graph management. Bing's distributed in-memory graph database A1 [Bur+20] leverages high-speed Remote Direct Memory Access to maintain an evolving graph containing billions of vertices and edges. Linkedin's in-memory graph database [Car+19] enables low latency read and write operations to the graph and supports the representation of N-ary relationships in the graphs. Our dynamic graph data structure does not require support for N-ary relationships, and thus can be implemented in a simpler manner.

5.7 Conclusions

This chapter presented the *Graph Feature Preprocessor*, a software library for fast feature extraction from dynamically changing transaction graphs. To achieve fast feature extraction, our library leverages an in-memory dynamic multigraph representation as well as our fine-grained parallel subgraph enumeration algorithms. When applied to a proprietary AML dataset, our graph feature extraction library is capable of processing more than 100,000 transactions per second. Effectively, a batch of 128 transactions can be processed in only 1 ms, enabling our library to be used in real-time settings.

This chapter also showed that the graph-based features generated by our library can significantly improve the accuracy of gradient-boosting-based machine learning models. Using graph-based features extracted from a proprietary AML dataset, we improved the minority class F1 score of LightGBM by 64%. In addition, our graph-based features improved minority class F1 scores of gradient-boosting-based machine learning models by up to 43% on publicly available AML datasets and by up to 35% on a real-world phishing detection dataset extracted from Ethereum. Furthermore, our graph ML pipeline that uses LightGBM for inference achieves competitive accuracy with the graph-neural-network-based solutions while achieving higher throughput and lower latency of processing a single batch of transactions.

6 Conclusions and Future Work

6.1 Conclusions

The use of graph pattern mining algorithms has become increasingly prevalent in recent years due to their ability to extract hidden knowledge from graphs. This capability has facilitated the development of a wide range of applications in various fields. The extraction of subgraph patterns from graphs makes it possible to predict protein interactions [YZT14; Yu+06], predict molecular properties [Bou+23; Che+20b], and, as shown in Chapter 5, detect financial crime. The most efficient algorithms for extracting subgraph patterns from a graph perform recursive search and exploration [Joh77; ELS13; SL20; LSL06]. However, even these algorithms could suffer from long execution times because the number of potential patterns that exist in a graph can grow exponentially with a graph parameter (see Section 1.2). To enable these and other related applications, fast graph pattern mining algorithms are required.

Acceleration of graph pattern mining. In this thesis, we show that graph pattern mining algorithms can be effectively accelerated using the existing manycore CPUs. To achieve fast enumeration of graph patterns, we focus on scalable parallelisation of the state-of-the-art sequential graph pattern mining algorithms. However, due to the irregular nature of the graphs, it is challenging to predict how much work each software thread would perform, which could cause workload imbalance (see Section 1.2). This thesis tackles this problem for algorithms that enumerate two different types of patterns: maximal clique and simple cycle. The algorithms for the enumeration of these patterns present different parallelisation challenges and opportunities and we address in unique ways. Whereas maximal clique enumeration algorithms [ELS13; TTT06] perform vertex-set intersection operations, simple cycle enumeration algorithms [Joh77; RT75] do not perform such computationally intensive operations. Furthermore, compared to simple cycle enumeration algorithms, the recursion trees of maximal clique enumeration algorithms are easier to decompose into fine-grained tasks, which facilitates their scalable parallelisation.

In Chapter 3, we address the challenges related to the acceleration of maximal clique enumeration [ELS13; TTT06] and introduce our fast parallel algorithm for this problem. The problem of

computationally intensive vertex-set intersection operations in that algorithm is investigated from both theoretical and practical perspectives. We theoretically show that a hash-join-based set intersection implementation leads to the maximal clique enumeration algorithm with lower theoretical time complexity compared to the algorithm that uses merge-join-based set intersections. For this reason, we implement a vectorised hash-join-based set intersection algorithm using hopscotch hashing [HST08]. Our set intersection implementation is faster than the alternatives when considering the cases that commonly occur in maximal clique enumeration algorithms. To achieve a scalable parallel implementation of this maximal clique enumeration algorithm that does not suffer from workload imbalance, we divide the execution of this algorithm into fine-grained tasks and employ a parallel processing framework with dynamic load balancing [VAR19]. As a result, our implementation scales almost linearly with the number of CPU cores and is, on average, an order of magnitude faster than the prior sequential and parallel maximal clique enumeration algorithms. Thus, we show that, by accelerating set intersection operations and employing fine-grained parallelism, maximal clique enumeration can be effectively accelerated using a manycore CPU.

Chapter 4 focusses on accelerating simple cycle enumeration [Joh77; RT75], which has a different set of challenges compared to the maximal clique enumeration. Because of the limited opportunities to exploit data parallelism, the main method used for acceleration is scalable parallelisation achieved by decomposing the recursion trees of the algorithms into fine-grained tasks. We demonstrate that such scalable parallelisation is possible even for the asymptotically-optimal simple cycle enumeration algorithm by Johnson [Joh77], despite the fact that the performance of this algorithm is strictly dependent on the order in which its recursive calls are executed. Although our fine-grained parallelisation of the Johnson algorithm performs more work in theory compared to the sequential execution of this algorithm, this additional work is insignificant in practice. Thus, our fine-grained parallelisation of the Johnson algorithm scales almost linearly in practice. However, if theoretical work-efficiency is also required, our fine-grained parallelisation of the Read-Tarjan algorithm [RT75] is available, which, thanks to our algorithmic optimisation, is able to achieve performance comparable to that of our fine-grained parallel Johnson algorithm. As a result, when executed on a system of four CPUs with 256 cores in total, both of our fine-grained parallel algorithms are able to achieve, on average, an order of magnitude speedup compared to the algorithms that use straightforward coarse-grained parallelisation method.

The acceleration methods presented in Chapters 3 and 4 are also applicable to other graph pattern mining algorithms. Chapter 4 shows that our method for fine-grained parallelisation of the Johnson algorithm can also be used to accelerate the search for temporal [KC18] and hop-constrained cycles [Pen+19]. In addition, the methods presented in Chapter 3 can be used to accelerate other graph pattern mining algorithms, such as biclique enumeration [LSL06], k-clique listing [DBS18], subgraph matching [SL20], and motif counting [MW19]. These algorithms rely on set operations, as illustrated in Figure 1.5, and can benefit from fast vectorised hash-join-based set operations. Furthermore, these algorithms also rely on recursive search, and their recursive calls can be executed in any order, which enables parallelising these algo-

rithms in a scalable manner. Thus, our fast parallel algorithms for maximal clique and simple cycle enumeration imply that other graph pattern mining algorithms can also be efficiently accelerated using manycore CPUs.

Application to financial crime detection. In addition to the acceleration of graph pattern mining algorithms, this thesis shows how these algorithms can be used for real-world applications, such as financial crime detection. The motivation for this part of the work is the observation that financial crime, such as money laundering and stock market manipulation, can manifest itself in the form of subgraph patterns within financial graphs, as illustrated in Figure 1.2. To facilitate financial crime detection tasks, we develop the Graph Feature Preprocessor library, presented in Chapter 4. This library is capable of enumerating known financial crime patterns in dynamically changing transaction graphs using fast graph mining algorithms, such as our parallel simple cycle enumeration algorithms introduced in Chapter 4. Our Graph Feature Preprocessor library encodes the enumerated graph patterns and various vertex statistics into the feature vector of incoming financial transactions. This enriched feature vector can be used to significantly boost the accuracy of financial crime detection tasks. We show that the graph-based features generated by our library increase the minority-class F1 score of gradient-boosting-based machine learning models [Ke+17; CG16] by up to 64% for the problem of money laundering detection and up to 35% for the problem of phishing detection. Furthermore, our Graph Feature Preprocessor library is capable of processing a batch of 128 transactions between 1 ms and 15 ms, enabling the applicability of our library in real-time settings.

6.2 Future Work

Distributed execution of graph pattern mining algorithms. To further accelerate graph pattern mining algorithms, the execution of these algorithms can be distributed across several manycore CPUs. Such a distributed system would provide more computational power and memory compared to a single CPU. In Section 4.6, we evaluated our cycle enumeration algorithms on a system consisting of several CPUs (see Table 4.4). However, no method was used for load balancing across CPUs, and the memory of each CPU contained a copy of the input graph. As a result, this type of distributed execution would not scale with the size of the graph or the number of CPUs used. To fully utilise the potential of a distributed system consisting of several CPUs, both computational power and memory of each CPU have to be efficiently used.

To enumerate subgraph patterns in larger graphs that could not fit in the memory of a single CPU, a graph could be divided into several partitions, and each CPU could locally process a partition assigned to it [KK98; Bad99; Che+16], as illustrated in Figure 6.1. However, if the generated partitions do not overlap, subgraph patterns that span across multiple partitions will not be found. Although this problem has been addressed in the case of clique enumeration [Che+16], partitioning the graph for the purpose of scalable distributed enumeration

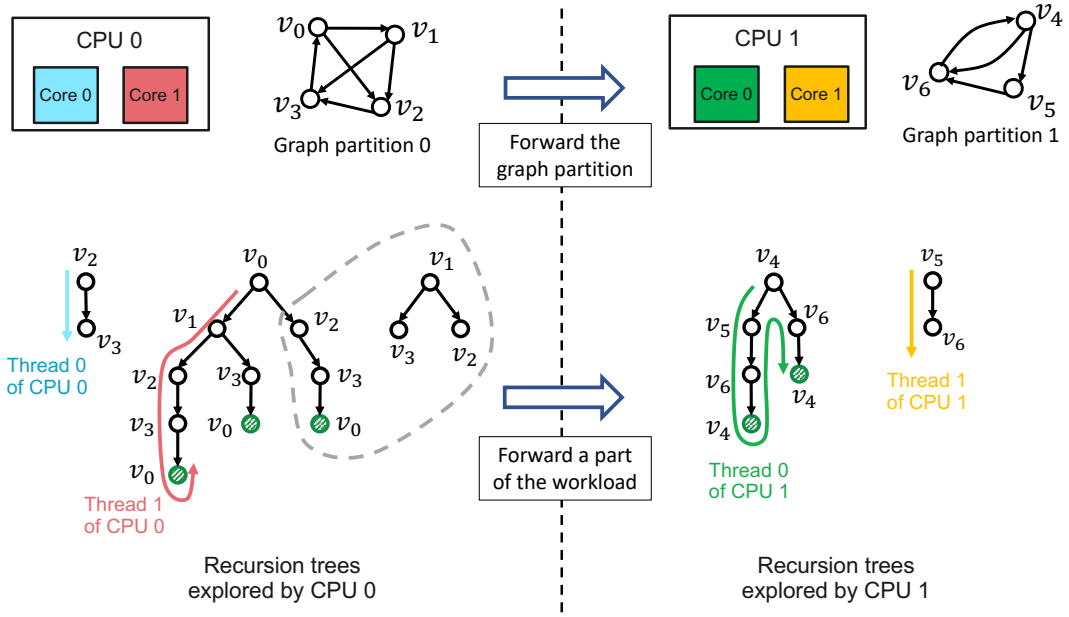


Figure 6.1: Distributing the execution of the cycle enumeration shown in Figure 1.3 using two CPUs with two cores per CPU. The graph from Figure 1.3a is divided into two partitions, and each partition is assigned to a CPU. To balance the workload across the CPUs, a portion of the workload indicated with a dashed line that CPU 0 executes can be forwarded to CPU 1. However, in this distributed setting, graph partition 0 would also have to be forwarded to CPU 1, which increases the network traffic.

of other patterns, such as simple cycles, is still an open research question. Furthermore, to dynamically balance the workload across CPUs (e.g., using Ray [Mor+18]), the entire graph partition would have to be moved between CPUs, as shown in Figure 6.1, resulting in increased network traffic and limited scalability. Thus, more work is required to address the aforementioned challenges and enable a scalable execution of our fast algorithms for maximal clique and simple cycle enumeration algorithms on distributed systems.

Custom hardware architecture for the acceleration of graph pattern mining. Another method for the further acceleration of graph pattern mining algorithms is hardware acceleration. For that purpose, one could use an existing platform, such as a GPU [Che+20a; CA22; JMV20; Guo+20], or develop a custom hardware accelerator [Yao+20; Che+21; Tal+22]. Existing GPU-based systems and custom hardware accelerators for graph pattern mining [Che+20a; CA22; JMV20; Guo+20; Yao+20; Che+21; Tal+22] support the execution of various mining algorithms, such as k-clique listing, motif counting, subgraph matching, and frequent subgraph mining. However, the abstraction used by these systems to make them user-friendly and easy to program prevents the implementation of fast algorithms specialised for each problem, such as the Bron-Kerbosch algorithm [ELS13; TTT06; BK73] for maximal clique enumeration, the Johnson algorithm [Joh77] for simple cycle enumeration, the solution by Sun and Luo [SL20] for subgraph matching, and the MineLMBC algorithm [LSL06] for biclique

Algorithm 16: The common approach for fine-grained parallelisation of different graph pattern mining algorithms [Joh77; LSL06; ELS13; SL20].

Input: SG - the current subgraph that mathces a portion of a pattern
D - auxiliary data structures
$\mathcal{G}$ - the input graph
InOut: T_1 - the thread that created this task

```
1  Task RecursiveCall (SG, D, G, T1)
2      T2 = the thread executing this task;
3      if T1 ≠ T2 then
4          copy_on_steal(D, T1, T2);                          ▷ see Section 4.3.2
5      if SG matches the pattern then
6          output SG;
7          return ;
8      D = update_start(SG, D, G);
9      cand = get_candidates(SG, D, G);          ▷ E.g., compute pivot for the BK algorithm
10     foreach c : cand do
11         if filter(c, SG, D, G) == true then
12             SG, D = update_spawn(c, SG, D, G);              ▷ E.g., insert c into SG
13             spawn RecursiveCall(SG, D, G, T2);
14     sync;
15     D = update_sync(SG, D, G);     ▷ E.g., update Blk and Blist of the Johnson algorithm
16 Function OuterLoop (G)
17     preprocessing(G);        ▷ E.g., degeneracy ordering of V in G for the BK algorithm
18     cand = get_global_candidates(G);                    ▷ E.g., all vertices V of G
19     parallel foreach c : cand do
20         T0 = the thread executing this loop iteration;
21         D = init(c, G);
22         spawn RecursiveCall({c}, D, G, T0);
23     sync;
```

enumeration. Furthermore, because these systems usually focus on enumerating patterns of fixed sizes, they cannot easily find patterns whose size can vary, such as maximal cliques, maximal bicliques, scatter-gather patterns, and simple cycles.

Based on the observations made in this thesis, a custom hardware accelerator for graph pattern mining should have the following properties: *(i)* support for fast set operations, *(ii)* dynamic load balancing across execution cores, and *(iii)* a programming model that enables the implementation of the existing state-of-the-art algorithms for graph pattern mining. The first property is required, given that various graph pattern mining algorithms rely on set operations [Cor+04; HLL13; SL20; MW19; LSL06; Che+22; DBS18; TTT06; ELS13; BK73], which can be a dominant part of their execution time [HZY18], as shown in Chapter 3. The second property is required to enable the scalable parallelisation of graph pattern algorithms using

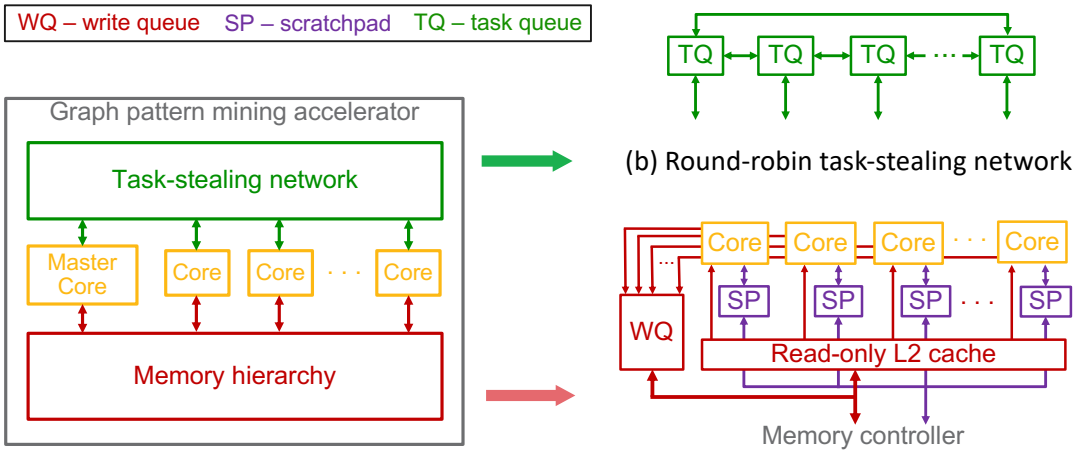


Figure 6.2: High-level overview of a manycore graph pattern mining accelerator. Each core supports fast set intersection operations and executes a *RecursionCall* task from Algorithm 16. The master core executes the *OuterLoop* function from the same algorithm, which generates the initial set of tasks. The task-stealing network enables workload balance across cores using work-stealing [BL99]. The memory hierarchy does not need to support hardware-managed cache coherency.

the fine-grained approach discussed in Chapters 3 and 4. This technique enables speeding up algorithms by designing a hardware accelerator with a higher number of execution cores. The programming model required by the third property ensures that the fastest graph pattern mining algorithms [Joh77; LSL06; ELS13; SL20] can be accelerated on such a custom hardware platform. Such a programming model is possible because these algorithms can fit the common structure shown in Algorithm 16, which can be parallelised in a fine-grained manner. Note that our algorithms presented in Chapters 3 and 4, as well as other graph pattern mining algorithms, such as MineLMBC [LSL06] and Sun and Luo [SL20], can be reformulated to fit this structure by redefining the highlighted functions of Algorithm 16.

A high-level overview of a potential graph pattern mining accelerator that supports all three of the aforementioned properties is shown in Figure 6.2a. In contrast to the out-of-order cores used by modern CPUs, the cores of our accelerator can be designed in a lightweight manner because they are only required to execute a fixed structure of the *RecursiveCall* task shown in Algorithm 16. Thus, in theory, our accelerator can contain more cores in the same area, which can further accelerate the execution of graph pattern mining algorithms. For this reason, the task-stealing network and memory hierarchy of our accelerator should be designed in such a way that their area scales with the number of cores. To achieve this behaviour, we propose the use of a round-robin task-stealing network shown in Figure 6.2b. The area of the round-robin task-stealing network scales better with the number of cores compared to a crossbar task-stealing network [Che+18] because the round-robin network does not require a direct connection between each pair of task queues. Additionally, it is challenging to scale the performance and area of the memory hierarchy that supports hardware-managed cache

coherency with the number of cores [Wan+20b; Fer+11; FNW15]. As a result, for our graph pattern mining accelerator, a custom memory hierarchy that does not require hardware-managed cache coherency, such as the one shown in Figure 6.2c, is preferable.

Our custom memory hierarchy illustrated in Figure 6.2c that does not require hardware-managed cache coherency is motivated by the following observations: *(i)* the input graph in the graph pattern mining algorithms [Joh77; RT75; ELS13; LSL06] is usually read-only; *(ii)* once these algorithms discover a subgraph, this subgraph will not be used again; *(iii)* our copy-on-steal mechanism, introduced in Section 4.3.2 and used by our programming model shown in Algorithm 16, enables each thread that executes one of our fine-grained algorithms presented in Chapter 4 to maintain its own set of data structures; *(iv)* the data structures of one thread are accessed by another thread only when a task stealing occurs; and *(v)* these data structures could fit in the on-chip memory of a processor (see Chapter 3). Observation *(i)* enables our accelerator to access the input graph using a read-only L2 cache and to use mechanisms for hiding cache-miss latencies [AI19b; AI19a; AI22] that could improve its performance. Observation *(ii)* enables the accelerator to asynchronously write the results to the main memory using a queue (see Figure 6.2c), which can then be directly written to the disc to prevent thrashing the main memory. As a result of *(iii)*, *(iv)*, and *(v)*, each core of our accelerator can have a dedicated scratchpad that could fit all the auxiliary data structures it requires, such as R, P, and X of the BK algorithm (see Algorithm 1) and Π, *Blk*, and *Blist* of the Johnson algorithm (see Algorithm 4). These data structures are transferred between scratchpads only when task stealing occurs, which can be accomplished by moving them through the main memory. Thus, each core only accesses the data from its own scratchpad. As a result, our custom memory hierarchy is not required to support a hardware-managed cache coherence protocol, which could enable designing the graph pattern mining accelerator with more cores.

Support for other subgraph patterns in the Graph Feature Preprocessor. In Chapter 5, we introduced our Graph Feature Preprocessor library that can generate features for financial transactions by extracting simple cycles, scatter-gather patterns, and fan-in/fan-out patterns, which are used to improve the accuracy of financial crime detection tasks. The feature space can be further increased by generating features based on other subgraph patterns, such as cliques [BK73], bicliques [LSL06], and user-defined subgraph patterns [SL20]. For this purpose, fast algorithms for the enumeration of these patterns are required, such as our parallel algorithm for maximal cycle enumeration, introduced in Chapter 3. To enable the integration of our parallel maximal cycle enumeration algorithm into the Graph Feature Preprocessor library, the algorithm has to be modified to support the processing of directed temporal multigraphs that are maintained by our library. The potential future research avenue is the development of even faster algorithms for the enumeration of the aforementioned patterns and the evaluation of their impact on the accuracy of financial crime detection tasks.

Automatic discovery of financial crime patterns. The Graph Feature Preprocessor presented in Chapter 5 relies on extracting known financial crime patterns, such as those shown in Fig-

ure 1.2. However, to avoid getting caught, criminals may develop more sophisticated financial crime patterns that our Graph Feature Preprocessor library cannot detect. To counteract this problem, a method that automatically learns new financial crime patterns is required. Graph Neural Networks (GNNs) represent the perfect candidate for this purpose because of their ability to operate on and learn from relational data. In addition, it has been shown that GNNs can count certain subgraph patterns in graphs and detect which subgraph patterns are relevant [Che+20b]. Further study is required to determine whether such an approach can be tailored to automatically detect the existing financial crime patterns in temporal multigraphs and discover new types of such patterns.

6.3 Final Remarks

The growth of data in recent years has necessitated the development of sophisticated algorithms capable of processing and analysing this vast and diverse data. The complexity of these algorithms is a consequence of the need to extract meaningful insights and patterns from graphs representing this data. As a result of their complexity, these algorithms may require a significant amount of time to execute and should thus be accelerated. This thesis demonstrates the feasibility of accelerating a subset of such graph algorithms as well as their the applicability to a real-world problem.

Bibliography

[Abd+16] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. "ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT, USA: IEEE, Nov. 2016, pp. 716–727. ISBN: 978-1-4673-8815-3. DOI: 10.1109/SC.2016.60.

[Abs17] Abseil. *Abseil Swiss Tables*. 2017. URL: https://abseil.io/blog/20180927-swisstables (visited on 05/08/2023).

[AFK97] Helmut Alt, Ulrich Fuchs, and Klaus Kriegel. "On the number of simple cycles in planar graphs". In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Rolf H. Möhring. Vol. 1335. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 15–24. ISBN: 978-3-540-63757-8. DOI: 10.1007/BFb0024484. URL: http://link.springer.com/10.1007/BFb0024484 (visited on 10/06/2021).

[Ahm+15] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. "Efficient Graphlet Counting for Large Networks". In: *2015 IEEE International Conference on Data Mining*. Atlantic City, NJ, USA: IEEE, Nov. 2015, pp. 1–10. ISBN: 978-1-4673-9504-5. DOI: 10.1109/ICDM.2015.141.

[AI19a] Mikhail Asiatici and Paolo Ienne. "DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Barcelona, Spain: IEEE, Sept. 2019, pp. 254–262. ISBN: 978-1-72814-884-7. DOI: 10.1109/FPL.2019.00049. URL: https://ieeexplore.ieee.org/document/8892073/ (visited on 10/12/2020).

[AI19b] Mikhail Asiatici and Paolo Ienne. "Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs". en. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA: ACM, Feb. 2019, pp. 310–319. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293901. URL: https://dl.acm.org/doi/10.1145/3289602.3293901 (visited on 10/12/2020).

[AI22] Mikhail Asiatici and Paolo Ienne. “Request, Coalesce, Serve, and Forget: Miss-Optimized Memory Systems for Bandwidth-Bound Cache-Unfriendly Applications on FPGAs”. en. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.2 (June 2022), pp. 1–33. ISSN: 1936-7406, 1936-7414. DOI: 10.1145/3466823. URL: https://dl.acm.org/doi/10.1145/3466823 (visited on 05/08/2023).

[All85] Eric W. Allender. “On the number of cycles possible in digraphs with large girth”. en. In: *Discrete Applied Mathematics* 10.3 (Mar. 1985), pp. 211–225. ISSN: 0166218X. DOI: 10.1016/0166-218X(85)90044-7. URL: https://linkinghub.elsevier.com/retrieve/pii/0166218X85900447 (visited on 03/30/2023).

[Alm+22] Mohammad Almasri, Yen-Hsiang Chang, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. *Parallelizing Maximal Clique Enumeration on GPUs*. arXiv:2212.01473 [cs]. Dec. 2022. URL: http://arxiv.org/abs/2212.01473 (visited on 12/08/2022).

[Alm22] Mohammad Almasri. “Accelerating graph pattern mining algorithms on modern graphics processing units”. PhD thesis. University of Illinois at Urbana-Champaign, May 2022.

[Alt21] Erik Altman. *AML-Data*. 2021. URL: https://github.com/IBM/AML-Data (visited on 05/30/2022).

[Ama23] Amazon. *Amazon Fraud Detector*. Accessed: 2023-01-10. 2023. URL: https://aws.amazon.com/fraud-detector/.

[ANS09] Yuriy Arbitman, Moni Naor, and Gil Segev. “De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results”. In: *Automata, Languages and Programming*. Vol. 5555. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 107–118. ISBN: 978-3-642-02927-1. DOI: 10.1007/978-3-642-02927-1_11. URL: http://link.springer.com/10.1007/978-3-642-02927-1_11 (visited on 06/02/2020).

[ANS10] Yuriy Arbitman, Moni Naor, and Gil Segev. “Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation”. In: *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. Las Vegas, NV, USA: IEEE, Oct. 2010, pp. 787–796. ISBN: 978-1-4244-8525-3. DOI: 10.1109/FOCS.2010.80. URL: http://ieeexplore.ieee.org/document/5671351/ (visited on 06/05/2020).

[AR16] Udit Agarwal and Vijaya Ramachandran. “Finding k Simple Shortest Paths and Cycles”. In: *27th International Symposium on Algorithms and Computation (ISAAC 2016)*. Ed. by Seok-Hee Hong. Vol. 64. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 8:1–8:12. ISBN: 978-3-95977-026-2. DOI: 10.4230/LIPIcs.ISAAC.2016.8. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6783.

[AT08] R. E. L. Aldred and Carsten Thomassen. “On the maximum number of cycles in a planar graph”. en. In: *J. Graph Theory* 57.3 (Mar. 2008), pp. 255–264. ISSN: 03649024, 10970118. DOI: 10.1002/jgt.20290.

[Ave11] Ching Avery. “Giraph: Large-scale graph processing infrastructure on Hadoop”. In: *Proceedings of the Hadoop Summit. Santa Clara* 11.3 (2011), pp. 5–9.

[AW10] Charu C. Aggarwal and Haixun Wang, eds. *Managing and Mining Graph Data*. en. Vol. 40. Advances in Database Systems. Boston, MA: Springer US, 2010. ISBN: 978-1-4419-6045-0. DOI: 10.1007/978-1-4419-6045-0.

[Bad99] David A. Bader. *A Practical Parallel Algorithm for Cycle Detection in Partitioned Digraphs*. 1999. URL: https://digitalrepository.unm.edu/ece_rpts/45 (visited on 05/08/2023).

[BAI23] Jovan Blanuša, Kubilay Atasu, and Paolo Ienne. “Fast Parallel Algorithms for Enumeration of Simple, Temporal, and Hop-Constrained Cycles”. en. In: *ACM Transactions on parallel computing* (2023), to appear.

[Bal97] V K Balakrishnan. *Graph Theory*. New York, NY: McGraw-Hill Professional, Feb. 1997.

[Bar+21] Pablo Barceló, Floris Geerts, Juan L. Reutter, and Maksimilian Ryschkov. “Graph Neural Networks with Local Graph Parameters”. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 2021, pp. 25280–25293. URL: https://proceedings.neurips.cc/paper/2021/hash/d4d8d1ac7e00e9105775a6b660dd3cbb-Abstract.html (visited on 05/05/2023).

[Bat+15] Omar Batarfi, Radwa El Shawi, Ayman G. Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi, and Sherif Sakr. “Large scale graph processing systems: survey and an experimental evaluation”. en. In: *Cluster Comput* 18.3 (Sept. 2015), pp. 1189–1213. ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-015-0472-6. URL: http://link.springer.com/10.1007/s10586-015-0472-6 (visited on 04/03/2023).

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. “Hash, Displace, and Compress”. en. In: *Algorithms - ESA 2009*. Vol. 5757. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 682–693. ISBN: 978-3-642-04127-3. DOI: 10.1007/978-3-642-04128-0_61. URL: http://link.springer.com/10.1007/978-3-642-04128-0_61 (visited on 06/05/2020).

[BC19] Anna D. Broido and Aaron Clauset. “Scale-free networks are rare”. en. In: *Nat Commun* 10.1 (Dec. 2019), p. 1017. ISSN: 2041-1723. DOI: 10.1038/s41467-019-08746-5.

[BE19] Ioana O. Bercea and Guy Even. “Fully-Dynamic Space-Efficient Dictionaries and Filters with Constant Number of Memory Accesses”. In: *arXiv:1911.05060 [cs]* (Nov. 2019). arXiv: 1911.05060.

[Bel58] Richard Bellman. “On a routing problem”. en. In: *Quart. Appl. Math.* 16.1 (1958), pp. 87–90. ISSN: 0033-569X, 1552-4485. DOI: 10.1090/qam/102435. URL: https://www.ams.org/qam/1958-16-01/S0033-569X-1958-0102435-2/ (visited on 04/03/2023).

[Ben22] BentoML. *A faster way to ship your models to production*. 2022. URL: https://www.bentoml.com/ (visited on 03/03/2023).

[BGL16] Austin R. Benson, David F. Gleich, and Jure Leskovec. “Higher-order organization of complex networks”. In: *Science* 353.6295 (2016), pp. 163–166. DOI: 10.1126/science.aad9029. URL: https://www.science.org/doi/abs/10.1126/science.aad9029.

[BIA22] Jovan Blanuša, Paolo Ienne, and Kubilay Atasu. “Scalable Fine-Grained Parallel Cycle Enumeration Algorithms”. en. In: *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. Philadelphia PA USA: ACM, July 2022, pp. 247–258. ISBN: 978-1-4503-9146-7. DOI: 10.1145/3490148.3538585. (Visited on 07/18/2022).

[Bir+13] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. “Optimal Listing of Cycles and st-Paths in Undirected Graphs”. en. In: *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA: SIAM, Jan. 2013, pp. 1884–1896. DOI: 10.1137/1.9781611973105.134.

[BK73] Coen Bron and Joep Kerbosch. “Algorithm 457: finding all cliques of an undirected graph”. In: *Communications of the ACM* 16.9 (Sept. 1973), pp. 575–577. ISSN: 00010782. DOI: 10.1145/362342.362367.

[BL99] Robert D. Blumofe and Charles E. Leiserson. “Scheduling multithreaded computations by work stealing”. In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 00045411. DOI: 10.1145/324133.324234.

[Bla+20a] Jovan Blanuša, Radu Stoica, Paolo Ienne, and Kubilay Atasu. “Manycore clique enumeration with fast set intersections”. en. In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 2676–2690. ISSN: 2150-8097. DOI: 10.14778/3407790.3407853.

[Bla+20b] Jovan Blanuša, Radu Stoica, Paolo Ienne, and Kubilay Atasu. “Parallelizing Maximal Clique Enumeration on Modern Manycore Processors”. In: *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, Louisiana, USA, May 18-22, 2020*. IEEE, 2020, pp. 211–214. DOI: 10.1109/IPDPSW50202.2020.00047.

[Ble90] Guy E. Blelloch. *Vector models for data-parallel computing*. Artificial intelligence. Cambridge, Mass: MIT Press, 1990. ISBN: 978-0-262-02313-9.

[Blu+96a] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. “Cilk: An Efficient Multithreaded Runtime System”. en. In: *Journal of Parallel and Distributed Computing* 37.1 (Aug. 1996), pp. 55–69. ISSN: 07437315. DOI: 10.1006/jpdc.1996.0107.

[Blu+96b] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". en. In: *Journal of Parallel and Distributed Computing* 37.1 (Aug. 1996), pp. 55–69. ISSN: 07437315. DOI: 10.1006/jpdc.1996.0107. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731596901070 (visited on 08/16/2019).

[BM10] Guy E. Blelloch and Bruce M. Maggs. "Parallel Algorithms". In: *Algorithms and theory of computation handbook*. Chapman & Hall/CRC Applied Algorithms and Data Structures series. London, England: CRC Press, 2010. Chap. 25, pp. 25.1–25.40.

[Bou+23] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M. Bronstein. "Improving Graph Neural Network Expressivity via Subgraph Isomorphism Counting". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 45.1 (Jan. 2023), pp. 657–668. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2022.3154319. (Visited on 01/10/2023).

[BP16] Albert-László Barabási and Márton Pósfai. "Network science". In: Cambridge, United Kingdom: Cambridge University Press, 2016. Chap. The scale-free property. ISBN: 978-1-107-07626-6.

[BPZ13] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. "Practical perfect hashing in nearly optimal space". en. In: *Information Systems* 38.1 (Mar. 2013), pp. 108–131. ISSN: 03064379. DOI: 10.1016/j.is.2012.06.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0306437912000944 (visited on 06/05/2020).

[Bre74] Richard P. Brent. "The Parallel Evaluation of General Arithmetic Expressions". In: *J. ACM* 21.2 (Apr. 1974), pp. 201–206. ISSN: 00045411. DOI: 10.1145/321812.321815. URL: http://portal.acm.org/citation.cfm?doid=321812.321815 (visited on 01/21/2020).

[Bri+19] Assia Brighen, Hachem Slimani, Abdelmounaam Rezgui, and Hamamache Kheddouci. "Listing all maximal cliques in large graphs on vertex-centric model". en. In: *The Journal of Supercomputing* (Feb. 2019). ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-019-02770-4. URL: http://link.springer.com/10.1007/s11227-019-02770-4 (visited on 03/27/2019).

[Buc+07] Kevin Buchin, Christian Knauer, Klaus Kriegel, André Schulz, and Raimund Seidel. "On the Number of Cycles in Planar Graphs". en. In: *Computing and Combinatorics*. Ed. by Guohui Lin. Vol. 4598. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 97–107. ISBN: 978-3-540-73544-1. DOI: 10.1007/978-3-540-73545-8_12. URL: http://link.springer.com/10.1007/978-3-540-73545-8_12 (visited on 10/06/2021).

[Bur+20] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, et al. "A1: A Distributed In-Memory Graph Database". en. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

Portland OR USA: ACM, June 2020, pp. 329–344. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3386135. (Visited on 02/28/2023).

[CA22] Xuhao Chen and Arvind. "Efficient and Scalable Graph Pattern Mining on GPUs". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 857–877. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/chen (visited on 05/03/2023).

[Cao+19] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. "TitAnt: online real-time transaction fraud detection in Ant Financial". en. In: *Proc. VLDB Endow.* 12.12 (Aug. 2019), pp. 2082–2093. ISSN: 2150-8097. DOI: 10.14778/3352063.3352126. (Visited on 01/10/2023).

[Car+18] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. "Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 40.4 (Apr. 2018), pp. 804–818. ISSN: 0162-8828, 2160-9292. DOI: 10.1109/TPAMI.2017.2696940. URL: http://ieeexplore.ieee.org/document/7907163/ (visited on 08/05/2020).

[Car+19] Andrew Carter, Andrew Rodriguez, Yiming Yang, and Scott Meyer. "Nanosecond Indexing of Graph Data With Hash Maps and VLists". en. In: *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, June 2019, pp. 623–635. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3314044. URL: https://dl.acm.org/doi/10.1145/3299869.3314044 (visited on 02/20/2023).

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[CH06] Diane J Cook and Lawrence B Holder, eds. *Mining Graph Data*. en. Nashville, TN: John Wiley & Sons, Nov. 2006. ISBN: 9780471731900.

[Che+16] Qun Chen, Chao Fang, Zhuo Wang, Bo Suo, Zhanhuai Li, and Zachary G. Ives. "Parallelizing Maximal Clique Enumeration Over Graph Data". In: *Database Systems for Advanced Applications*. Vol. 9643. Cham: Springer International Publishing, 2016, pp. 249–264. ISBN: 978-3-319-32049-6. DOI: 10.1007/978-3-319-32049-6_16. URL: http://link.springer.com/10.1007/978-3-319-32049-6_16 (visited on 03/20/2019).

[Che+18] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. "An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 55–67. ISBN: 978-1-

5386-6240-3. DOI: 10.1109/MICRO.2018.00014. URL: https://ieeexplore.ieee.org/document/8574531/ (visited on 09/09/2020).

[Che+19a] Liang Chen, Jiaying Peng, Yang Liu, Jintang Li, Fenfang Xie, and Zibin Zheng. *XBLOCK Blockchain Datasets: InPlusLab Ethereum Phishing Detection Datasets*. http://xblock.pro/ethereum/. 2019.

[Che+19b] Xucan Chen, Mohammad Al Hasan, Xintao Wu, Pavel Skums, Mohammad Javad Feizollahi, Marie Ouellet, Eric L. Sevigny, David Maimon, and Yubao Wu. "Characteristics of Bitcoin Transactions on Cryptomarkets". en. In: *Security, Privacy, and Anonymity in Computation, Communication, and Storage*. Ed. by Guojun Wang, Jun Feng, Md Zakirul Alam Bhuiyan, and Rongxing Lu. Vol. 11611. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 261–276. ISBN: 978-3-030-24906-9. DOI: 10.1007/978-3-030-24907-6_20. URL: http://link.springer.com/10.1007/978-3-030-24907-6_20 (visited on 04/04/2023).

[Che+20a] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. "Pangolin: an efficient and flexible graph mining system on CPU and GPU". en. In: *Proc. VLDB Endow.* 13.10 (June 2020), pp. 1190–1205. ISSN: 2150-8097. DOI: 10.14778/3389133.3389137. URL: https://dl.acm.org/doi/10.14778/3389133.3389137 (visited on 08/29/2020).

[Che+20b] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. "Can Graph Neural Networks Count Substructures?" In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/75877cb75154206c4e65e76b88a12712-Abstract.html.

[Che+21] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. "FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, June 2021, pp. 581–594. ISBN: 978-1-66543-333-4. DOI: 10.1109/ISCA52012.2021.00052. URL: https://ieeexplore.ieee.org/document/9499844/ (visited on 05/05/2023).

[Che+22] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu, and Jianxin Li. "Efficient maximal biclique enumeration for large sparse bipartite graphs". en. In: *Proc. VLDB Endow.* 15.8 (Apr. 2022), pp. 1559–1571. ISSN: 2150-8097. DOI: 10.14778/3529337.3529341. URL: https://dl.acm.org/doi/10.14778/3529337.3529341 (visited on 03/30/2023).

[CK08] F. Cazals and C. Karande. "A note on the problem of reporting maximal cliques". en. In: *Theoretical Computer Science* 407.1-3 (Nov. 2008), pp. 564–568. ISSN: 03043975. DOI: 10.1016/j.tcs.2008.05.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397508003903 (visited on 07/02/2020).

[Con+16] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. “Sublinear-Space Bounded-Delay Enumeration for Massive Network Analytics: Maximal Cliques”. en. In: (2016), 15 pages. DOI: 10.4230/LIPICS.ICALP.2016.148. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6292/ (visited on 05/27/2020).

[Con63] Melvin E. Conway. “A multiprocessor system design”. en. In: *Proceedings of the November 12-14, 1963, fall joint computer conference on XX - AFIPS '63 (Fall)*. Las Vegas, Nevada: ACM Press, 1963, p. 139. DOI: 10.1145/1463822.1463838. URL: http://portal.acm.org/citation.cfm?doid=1463822.1463838 (visited on 03/25/2023).

[Cor+04] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. “A (sub)graph isomorphism algorithm for matching large graphs”. en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (Oct. 2004), pp. 1367–1372. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.75. URL: http://ieeexplore.ieee.org/document/1323804/ (visited on 03/04/2019).

[Cor+20] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. “Principal Neighbourhood Aggregation for Graph Nets”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 13260–13271. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/99cad265a1768cc2dd013f0e740300ae-Paper.pdf.

[Cor09a] “Introduction to algorithms”. In: ed. by Thomas H. Cormen. 3rd ed. OCLC: ocn311310321. Cambridge, Mass: MIT Press, 2009. Chap. Graph Algorithms. ISBN: 978-0-262-53305-8.

[Cor09b] “Introduction to algorithms”. In: ed. by Thomas H. Cormen. 3rd ed. Cambridge, Mass: MIT Press, 2009. Chap. 11. ISBN: 978-0-262-03384-8.

[Cor23] Livio Corselli. “Italy: money transfer, money laundering and intermediary liability”. en. In: *JFC* 30.2 (Feb. 2023), pp. 377–388. ISSN: 1359-0790, 1359-0790. DOI: 10.1108/JFC-10-2019-0137. URL: https://www.emerald.com/insight/content/doi/10.1108/JFC-10-2019-0137/full/html (visited on 04/04/2023).

[COR93] CORPORATE The MPI Forum. “MPI: a message passing interface”. en. In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*. Portland, Oregon, United States: ACM Press, 1993, pp. 878–883. ISBN: 978-0-8186-4340-8. DOI: 10.1145/169627.169855.

[CS20] Tao-Hung Chang and Davor Svetinovic. “Improving Bitcoin Ownership Identification Using Transaction Patterns Analysis”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 50.1 (2020), pp. 9–20. DOI: 10.1109/TSMC.2018.2867497.

[CSB22] Mário Cardoso, Pedro Saleiro, and Pedro Bizarro. "LaundroGraph: Self-Supervised Graph Representation Learning for Anti-Money Laundering". In: *Proceedings of the Third ACM International Conference on AI in Finance*. 2022, pp. 130–138.

[Cse+22] Andras Cser, Merritt Maxix, Caroline Provost, and Peggy Dostie. *The Forrester Wave™: Anti-Money-Laundering Solutions, Q3 2022*. Tech. rep. Accessed: 2023-01-10. Forrester, 2022, pp. 1–10. URL: https://www.forrester.com/report/the-forrester-wave-tm-anti-money-laundering-solutions-q3-2022/RES176346.

[Cui+17] Huanqing Cui, Jian Niu, Chuanai Zhou, and Minglei Shu. "A Multi-Threading Algorithm to Detect and Remove Cycles in Vertex- and Arc-Weighted Digraph". en. In: *Algorithms* 10.4 (Oct. 2017), p. 115. ISSN: 1999-4893. DOI: 10.3390/a10040115.

[Dan+11] Leon Danon, Ashley Ford, Thomas House, Chris Jewell, Matt Keeling, Gareth Roberts, Joshua Ross, and Matthew Vernon. "Networks and the Epidemiology of Infectious Disease". In: *Interdisciplinary perspectives on infectious diseases* 2011 (Mar. 2011), p. 284909. DOI: 10.1155/2011/284909.

[Dan68] G. Danielson. "On finding the simple paths and circuits in a graph". In: *IEEE Trans. Circuit Theory* 15.3 (Sept. 1968), pp. 294–295. ISSN: 0018-9324. DOI: 10.1109/TCT.1968.1082837.

[DBS18] Maximilien Danisch, Oana Balalau, and Mauro Sozio. "Listing k-cliques in Sparse Real-World Graphs*". en. In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*. Lyon, France: ACM Press, 2018, pp. 589–598. ISBN: 978-1-4503-5639-8. DOI: 10.1145/3178876.3186125. URL: http://dl.acm.org/citation.cfm?doid=3178876.3186125 (visited on 03/30/2023).

[dev22a] scikit-learn developers. *Scikit-learn: Preprocessing Data*. Accessed: 2023-01-16. 2022. URL: https://scikit-learn.org/stable/modules/preprocessing.html.

[dev22b] scikit-learn developers. *Scikit-learn: StandardScaler*. Accessed: 2023-01-16. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[Die+97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. "A Reliable Randomized Algorithm for the Closest-Pair Problem". en. In: *Journal of Algorithms* 25.1 (Oct. 1997), pp. 19–51. ISSN: 01966774. DOI: 10.1006/jagm.1997.0873. URL: https://linkinghub.elsevier.com/retrieve/pii/S0196677497908737 (visited on 06/24/2020).

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". en. In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X, 0945-3245. DOI: 10.1007/BF01386390. URL: http://link.springer.com/10.1007/BF01386390 (visited on 04/03/2023).

[DM90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. “A new universal class of hash functions and dynamic hashing in real time”. en. In: *Automata, Languages and Programming*. Ed. by Michael S. Paterson. Vol. 443. Series Title: Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1990, pp. 6–19. ISBN: 978-3-540-52826-5. DOI: 10.1007/BFb0032018. (Visited on 06/05/2020).

[doc23] Python docs. *pickle - Python object serialization*. Accessed: 2023-02-21. 2023. URL: https://docs.python.org/3/library/pickle.html.

[DST20] Apurba Das, Seyed-Vahid Sanei-Mehri, and Srikanta Tirthapura. “Shared-memory Parallel Maximal Clique Enumeration from Static and Dynamic Graphs”. en. In: *ACM Trans. Parallel Comput.* 7.1 (Apr. 2020), pp. 1–28. ISSN: 2329-4949, 2329-4957. DOI: 10.1145/3380936. URL: https://dl.acm.org/doi/10.1145/3380936 (visited on 05/12/2020).

[Edd+22] Ahmad Naser Eddin, Jacopo Bono, David Aparício, David Polido, João Tiago Ascensão, Pedro Bizarro, and Pedro Ribeiro. *Anti-Money Laundering Alert Optimization Using Machine Learning with Graphs*. en. arXiv:2112.07508 [cs]. June 2022. URL: http://arxiv.org/abs/2112.07508 (visited on 01/09/2023).

[Edi+12] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. “STINGER: High performance data structure for streaming graphs”. In: *2012 IEEE Conference on High Performance Extreme Computing*. Waltham, MA, USA: IEEE, Sept. 2012, pp. 1–5. DOI: 10.1109/HPEC.2012.6408680. URL: http://ieeexplore.ieee.org/document/6408680/ (visited on 02/21/2022).

[EG59] P. Erdős and T. Gallai. “On maximal paths and circuits of graphs”. en. In: *Acta mathematica Academiae Scientiarum Hungaricae* 10.3-4 (Sept. 1959), pp. 337–356. ISSN: 0001-5954, 1588-2632. DOI: 10.1007/BF02024498.

[Eks+18] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. “Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time”. In: *Proceedings of the 2018 World Wide Web Conference*. WWW ’18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1775–1784. ISBN: 9781450356398. DOI: 10.1145/3178876.3186183. URL: https://doi.org/10.1145/3178876.3186183.

[Els+14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. “GraMi: frequent subgraph and pattern mining in a single large graph”. en. In: *Proc. VLDB Endow.* 7.7 (2014), pp. 517–528. ISSN: 2150-8097. DOI: 10.14778/2732286.2732289. URL: http://dl.acm.org/doi/10.14778/2732286.2732289 (visited on 03/15/2020).

[ELS10] David Eppstein, Maarten Löffler, and Darren Strash. “Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time”. In: *Algorithms and Computation*. Ed. by Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park. Vol. 6506. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 403–414. ISBN: 978-3-642-17517-6. DOI: 10.1007/978-3-642-17517-6_36.

[ELS13] David Eppstein, Maarten Löffler, and Darren Strash. “Listing All Maximal Cliques in Large Sparse Real-World Graphs in Near-Optimal Time”. en. In: *J. Exp. Algorithmics* 18 (Dec. 2013), pp. 3.1–3.21. ISSN: 10846654. DOI: 10.1145/2543629. (Visited on 09/19/2019).

[Epp94] David Eppstein. “Arboricity and bipartite subgraph listing algorithms”. en. In: *Information Processing Letters* 51.4 (Aug. 1994), pp. 207–211. ISSN: 00200190. DOI: 10.1016/0020-0190(94)90121-X. URL: https://linkinghub.elsevier.com/retrieve/pii/002001909490121X (visited on 07/13/2021).

[ES12] David Eppstein and Emma S. Spiro. “The h-Index of a Graph and its Application to Dynamic Subgraph Statistics”. In: *JGAA* 16.2 (2012). arXiv: 0904.3741, pp. 543–567. ISSN: 1526-1719. DOI: 10.7155/jgaa.00273. URL: http://arxiv.org/abs/0904.3741 (visited on 01/16/2020).

[Fan+19] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. “Graph Neural Networks for Social Recommendation”. In: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. Ed. by Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia. ACM, 2019, pp. 417–426. DOI: 10.1145/3308558.3313488.

[Fer+11] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. “Cuckoo directory: A scalable directory for many-core systems”. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. San Antonio, TX, USA: IEEE, Feb. 2011, pp. 169–180. ISBN: 978-1-4244-9432-3. DOI: 10.1109/HPCA.2011.5749726. URL: http://ieeexplore.ieee.org/document/5749726/ (visited on 05/08/2023).

[FHP00] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. “On Identifying Strongly Connected Components in Parallel”. en. In: *Parallel and Distributed Processing*. Ed. by José Rolim. Vol. 1800. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 505–511. ISBN: 978-3-540-67442-9. DOI: 10.1007/3-540-45591-4_68. URL: http://link.springer.com/10.1007/3-540-45591-4_68 (visited on 11/16/2020).

[Fin09] Tony Finch. “Incremental calculation of weighted mean and variance”. In: (Jan. 2009), pp. 1–8.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. “Storing a Sparse Table with 0(1) Worst Case Access Time”. In: *J. ACM* 31.3 (June 1984), pp. 538–544. ISSN: 0004-5411. DOI: 10.1145/828.1884. URL: https://doi.org/10.1145/828.1884.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The implementation of the Cilk-5 multithreaded language”. en. In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*. Montreal, Quebec, Canada: ACM Press, 1998, pp. 212–223. ISBN: 978-0-89791-987-6. DOI: 10.1145/277650.277725. URL: http://portal.acm.org/citation.cfm?doid=277650.277725 (visited on 05/23/2020).

[FMG22] Per Fuchs, Domagoj Margan, and Jana Giceva. “Sortledton: a universal, transactional graph data structure”. en. In: *Proc. VLDB Endow.* 15.6 (Feb. 2022), pp. 1173–1186. ISSN: 2150-8097. DOI: 10.14778/3514061.3514065. URL: https://dl.acm.org/doi/10.14778/3514061.3514065 (visited on 03/03/2023).

[FNW15] Yaosheng Fu, Tri M. Nguyen, and David Wentzlaff. “Coherence domain restriction on large scale systems”. en. In: *Proceedings of the 48th International Symposium on Microarchitecture*. Waikiki Hawaii: ACM, Dec. 2015, pp. 686–698. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830832. URL: https://dl.acm.org/doi/10.1145/2830772.2830832 (visited on 05/08/2023).

[FO19] Pierre Fraigniaud and Dennis Olivetti. “Distributed Detection of Cycles”. en. In: *ACM Trans. Parallel Comput.* 6.3 (Dec. 2019), pp. 1–20. ISSN: 2329-4949, 2329-4957. DOI: 10.1145/3322811. (Visited on 01/29/2021).

[For10] Santo Fortunato. “Community detection in graphs”. In: *Physics Reports* 486.3-5 (Feb. 2010). arXiv: 0906.0612, pp. 75–174. ISSN: 03701573. DOI: 10.1016/j.physrep.2009.11.002. URL: http://arxiv.org/abs/0906.0612 (visited on 05/03/2022).

[Gau+21] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, et al. “Utilizing graph machine learning within drug discovery and development”. In: *Briefings in Bioinformatics* 22.6 (May 2021). ISSN: 1477-4054. DOI: 10.1093/bib/bbab159. URL: https://doi.org/10.1093/bib/bbab159.

[Gib69] Norman E. Gibbs. “A Cycle Generation Algorithm for Finite Undirected Linear Graphs”. en. In: *J. ACM* 16.4 (Oct. 1969), pp. 564–568. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321541.321545.

[GKL12] Serge Gaspers, Dieter Kratsch, and Mathieu Liedloff. “On Independent Sets and Bicliques in Graphs”. en. In: *Algorithmica* 62.3-4 (Apr. 2012), pp. 637–658. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/s00453-010-9474-1. URL: http://link.springer.com/10.1007/s00453-010-9474-1 (visited on 07/13/2021).

[Goo+12] Michael T. Goodrich, Daniel S. Hirschberg, Michael Mitzenmacher, and Justin Thaler. “Cache-Oblivious Dictionaries and Multimaps with Negligible Failure Probability”. In: *Design and Analysis of Algorithms*. Vol. 7659. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 203–218. ISBN: 978-3-642-34862-4. DOI: 10.1007/978-3-642-34862-4_15.

URL: http://link.springer.com/10.1007/978-3-642-34862-4_15 (visited on 06/05/2020).

[Goo22] Google Cloud. *General-purpose machine family: N1 machine series*. 2022. URL: https://cloud.google.com/compute/docs/general-purpose-machines (visited on 11/14/2022).

[GOV22] Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. "Why do tree-based models still outperform deep learning on typical tabular data?" In: *36th Conference on Neural Information Processing Systems (NeurIPS 2022) Track on Datasets and Benchmarks*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 507–520. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/0378c7692da36807bdec87ab043cdadc-Paper-Datasets_and_Benchmarks.pdf.

[Gro16] Roberto Grossi. "Enumeration of Paths, Cycles, and Spanning Trees". en. In: *Encyclopedia of Algorithms*. New York, NY: Springer New York, 2016, pp. 640–645. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4{_}728.

[GRW17] Pierre-Louis Giscard, Paul Rochet, and Richard C. Wilson. "Evaluating balance on social networks from their simple cycles". en. In: *Journal of Complex Networks* 5 (May 2017), pp. 750–775. ISSN: 2051-1310, 2051-1329. DOI: 10.1093/comnet/cnx005.

[GS05] A. Gupta and C. Selvidge. "Acyclic modeling of combinational loops". In: *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005*. San Jose, CA: IEEE, 2005, pp. 343–348. ISBN: 978-0-7803-9254-0. DOI: 10.1109/ICCAD.2005.1560091.

[GS21] Anshul Gupta and Toyotaro Suzumura. *Finding All Bounded-Length Simple Cycles in a Directed Graph*. May 2021. arXiv: 2105.10094 [cs.DS].

[Guo+20] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. "GPU-Accelerated Subgraph Enumeration on Partitioned Graphs". en. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, June 2020, pp. 1067–1082. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3389699. URL: https://dl.acm.org/doi/10.1145/3318464.3389699 (visited on 05/05/2023).

[HK20] László Hajdu and Miklós Krész. "Temporal Network Analytics for Fraud Detection in the Banking Sector". en. In: *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*. Vol. 1260. Series Title: Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 145–157. ISBN: 978-3-030-55814-7. DOI: 10.1007/978-3-030-55814-7_12. (Visited on 01/28/2021).

[HLL13] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. "TurboISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases". In: June 2013, pp. 337–348. DOI: 10.1145/2463676.2465300.

[HM20] Danny Hermelin and George Manoussakis. “Efficient enumeration of maximal induced bicliques”. en. In: *Discrete Applied Mathematics* (June 2020). Combined Special Issue: 1) 17th Cologne–Twente Workshop on Graphs and Combinatorial Optimization (CTW 2019); Guest edited by Johann Hurink, Bodo Manthey 2) WEPA 2018 (Second Workshop on Enumeration Problems and Applications); Guest edited by Takeaki Uno, Andrea Marino, pp. 253–261. ISSN: 0166218X. DOI: 10.1016/j.dam.2020.04.034. URL: https://linkinghub.elsevier.com/retrieve/pii/S0166218X20302365 (visited on 07/13/2021).

[Hou+16] Boyi Hou, Zhuo Wang, Qun Chen, Bo Suo, Chao Fang, Zhanhuai Li, and Zachary G. Ives. “Efficient Maximal Clique Enumeration Over Graph Data”. en. In: *Data Sci. Eng.* 1.4 (Dec. 2016), pp. 219–230. ISSN: 2364-1185, 2364-1541. DOI: 10.1007/s41019-017-0033-5. URL: http://link.springer.com/10.1007/s41019-017-0033-5 (visited on 09/18/2019).

[HST08] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. “Hopscotch Hashing”. en. In: *Distributed Computing*. Vol. 5218. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 350–364. ISBN: 978-3-540-87778-3. DOI: 10.1007/978-3-540-87779-0_24. URL: http://link.springer.com/10.1007/978-3-540-87779-0_24 (visited on 06/15/2020).

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. “Inductive Representation Learning on Large Graphs”. In: *NIPS*. 2017.

[HZY18] Shuo Han, Lei Zou, and Jeffrey Xu Yu. “Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions”. en. In: *Proceedings of the 2018 International Conference on Management of Data - SIGMOD ’18*. Houston, TX, USA: ACM Press, 2018, pp. 1587–1602. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3196924.

[IBM22a] IBM Research. *Graph Feature Preprocessor Public Examples*. Accessed: 2023-03-3. 2022. URL: https://github.com/IBM/snapml-examples/blob/main/examples/graph_feature_preprocessor/graph_feature_preprocessor.ipynb.

[IBM22b] IBM Research. *Graph Feature Preprocessor PyPI Documentation*. Accessed: 2023-01-10. 2022. URL: https://snapml.readthedocs.io/en/latest/graph_preprocessor.html.

[IBM22c] IBM Research. *Snap ML PyPI package*. Accessed: 2023-01-10. 2022. URL: https://pypi.org/project/snapml/.

[IBM23a] IBM. *Cloud Pak for Data*. Accessed: 2023-02-21. 2023. URL: https://www.ibm.com/products/cloud-pak-for-data.

[IBM23b] IBM. *z16*. 2023. URL: https://www.ibm.com/products/z16 (visited on 02/23/2023).

[IBM23c] IBM Research. *IBM Transactions for Anti Money Laundering (AML)*. Accessed: 2023-02-28. 2023. URL: https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml.

[IOT14] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. "Faster set intersection with SIMD instructions by reducing branch mispredictions". en. In: *Proc. VLDB Endow.* 8.3 (2014), pp. 293–304. ISSN: 21508097. DOI: 10.14778/2735508.2735518.

[Isl+09] Md. Nazrul Islam, S. M. Rafizul Haque, Kaji Masudul Alam, and Md. Tarikuzzaman. "An approach to improve collusion set detection using MCL algorithm". In: *2009 12th International Conference on Computers and Information Technology*. Dhaka, Bangladesh: IEEE, Dec. 2009, pp. 237–242. ISBN: 978-1-4244-6281-0. DOI: 10.1109/ICCIT.2009.5407133.

[JaJ92] Joseph JaJa. *Introduction to parallel algorithms*. en. Boston, MA: Addison Wesley, Mar. 1992.

[JCZ13] Chuntao Jiang, Frans Coenen, and Michele Zito. "A survey of frequent subgraph mining algorithms". en. In: *The Knowledge Engineering Review* 28.1 (Mar. 2013), pp. 75–105. ISSN: 0269-8889, 1469-8005. DOI: 10.1017/S0269888912000331. URL: https://www.cambridge.org/core/product/identifier/S0269888912000331/type/journal_article (visited on 04/03/2023).

[Jia+13] Zhi-Qiang Jiang, Wen-Jie Xie, Xiong Xiong, Wei Zhang, Yong-Jie Zhang, and Wei-Xing Zhou. "Trading networks, abnormal motifs and stock manipulation". en. In: *Quantitative Finance Letters* 1.1 (Dec. 2013), pp. 1–8. ISSN: 2164-9502, 2164-9510. DOI: 10.1080/21649502.2013.802877.

[JMB17] Jaroslaw Jankowski, Radosław Michalski, and Piotr Bródka. *Spreading processes in multilayer complex network within virtual world*. 2017. DOI: 10.7910/DVN/V6AJRV.

[JMV20] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. "Peregrine: a pattern-aware graph mining system". en. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. Heraklion Greece: ACM, Apr. 2020, pp. 1–16. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387548. (Visited on 12/27/2021).

[JN14] Kevin Jamieson and Robert Nowak. "Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting". In: *2014 48th Annual Conference on Information Sciences and Systems (CISS)*. Princeton, NJ, USA: IEEE, Mar. 2014, pp. 1–6. ISBN: 978-1-4799-3001-2. DOI: 10.1109/CISS.2014.6814096. URL: http://ieeexplore.ieee.org/document/6814096/ (visited on 05/08/2023).

[Joh75] Donald B. Johnson. "Finding All the Elementary Circuits of a Directed Graph". en. In: *SIAM J. Comput.* 4.1 (Mar. 1975), pp. 77–84. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0204007.

[Joh77] Donald B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". en. In: *J. ACM* 24.1 (Jan. 1977), pp. 1–13. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321992.321993. URL: https://dl.acm.org/doi/10.1145/321992.321993 (visited on 04/03/2023).

[JS19] Wole Jaiyeoba and Kevin Skadron. “GraphTinker: A High Performance Data Structure for Dynamic Graph Processing”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil: IEEE, May 2019, pp. 1030–1041. ISBN: 978-1-72811-246-6. DOI: 10.1109/IPDPS.2019.00110. (Visited on 02/21/2022).

[Kam67] T. Kamae. “A Systematic Method of Finding All Directed Circuits and Enumerating All Directed Paths”. In: *IEEE Trans. Circuit Theory* 14.2 (June 1967), pp. 166–171. ISSN: 0018-9324. DOI: 10.1109/TCT.1967.1082699.

[Kan+22] Hiroki Kanezashi, Toyotaro Suzumura, Xin Liu, and Takahiro Hirofuchi. *Ethereum Fraud Detection with Heterogeneous Graph Neural Networks*. arXiv:2203.12363 [cs]. July 2022. URL: http://arxiv.org/abs/2203.12363 (visited on 03/03/2023).

[KC07] Yung-Keun Kwon and Kwang-Hyun Cho. “Analysis of feedback loops and robustness in network evolution based on Boolean models”. en. In: *BMC Bioinformatics* 8 (2007), p. 430. ISSN: 1471-2105. DOI: 10.1186/1471-2105-8-430.

[KC18] Rohit Kumar and Toon Calders. “2SCENT: an efficient algorithm for enumerating all simple temporal cycles”. en. In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1441–1453. ISSN: 2150-8097. DOI: 10.14778/3236187.3236197.

[Ke+17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. “Lightgbm: A highly efficient gradient boosting decision tree”. In: *Advances in neural information processing systems* 30 (2017), pp. 3146–3154.

[KK09] Steffen Klamt and Axel von Kamp. “Computing paths and cycles in biological interaction graphs”. en. In: *BMC Bioinformatics* 10.1 (Dec. 2009), p. 181. ISSN: 1471-2105. DOI: 10.1186/1471-2105-10-181.

[KK98] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

[KM11] Nancy Kinnison and John Madinger, eds. *Money Laundering: A Guide for Criminal Investigators, Third Edition*. en. Boston, MA: Routledge, 2011. ISBN: 978-1439869123.

[Knu68] Donald Ervin Knuth. *The art of computer programming. Volume 1, Volume 1,* English. OCLC: 489816776. 1968. ISBN: 978-0-201-03801-9.

[Kuk07] Alexey Kukanov. “The Foundations for Scalable Multicore Software in Intel Threading Building Blocks”. In: *ITJ* 11.04 (Nov. 2007). ISSN: 1535864X, 1535864X. DOI: 10.1535/itj.1104.05. URL: http://www.intel.com/technology/itj/2007/v11i4/5-foundations/1-abstract.htm (visited on 07/03/2019).

[Kun13] Jérôme Kunegis. “KONECT: the Koblenz network collection”. en. In: *Proceedings of the 22nd International Conference on World Wide Web*. Rio de Janeiro, Brazil: ACM Press, 2013, pp. 1343–1350. ISBN: 978-1-4503-2038-2. DOI: 10.1145/2487788.2488173.

[KW17] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[KZ00] Stephen Kokoska and Daniel Zwillinger. *CRC Standard Probability and Statistics Tables and Formulae, Student Edition*. en. 0th ed. CRC Press, Mar. 2000. ISBN: 978-0-429-18146-7. DOI: 10.1201/b16923. (Visited on 01/24/2023).

[LBK16] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. “SIMD Compression and the Intersection of Sorted Integers”. In: *Softw. Pract. Exper.* 46.6 (June 2016). arXiv: 1401.6399, pp. 723–749. ISSN: 00380644. DOI: 10.1002/spe.2326. URL: http://arxiv.org/abs/1401.6399 (visited on 02/25/2020).

[Lee+10] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. “Using memory mapping to support cactus stacks in work-stealing runtime systems”. en. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT ’10*. Vienna, Austria: ACM Press, 2010, p. 411. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854324. URL: http://portal.acm.org/citation.cfm?doid=1854273.1854324 (visited on 05/23/2020).

[Lee+20] Meng-Chieh Lee, Yue Zhao, Aluna Wang, Pierre Jinghong Liang, Leman Akoglu, Vincent S. Tseng, and Christos Faloutsos. “AutoAudit: Mining Accounting and Time-Evolving Graphs”. In: *2020 IEEE International Conference on Big Data (Big Data)*. Atlanta, GA, USA: IEEE, Dec. 2020, pp. 950–956. ISBN: 978-1-72816-251-5. DOI: 10.1109/BigData50022.2020.9378346. URL: https://ieeexplore.ieee.org/document/9378346/ (visited on 04/03/2023).

[Les+17] Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs, and E. Wes Bethel. “Maximal clique enumeration with data-parallel primitives”. en. In: *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. Phoenix, AZ: IEEE, Oct. 2017, pp. 16–25. ISBN: 978-1-5386-0617-9. DOI: 10.1109/LDAV.2017.8231847. URL: http://ieeexplore.ieee.org/document/8231847/ (visited on 02/25/2019).

[Li+20] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. “FlowScope: Spotting Money Laundering Based on Graphs”. In: *AAAI* 34.04 (Apr. 2020), pp. 4731–4738. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v34i04.5906. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5906 (visited on 04/03/2023).

[Lic+22] Cedric Lichtenau, Alper Buyuktosunoglu, Ramon Bertran, Peter Figuli, Christian Jacobi, Nikolaos Papandreou, Haris Pozidis, Anthony Saporito, Andrew Sica, and Elpida Tzortzatos. "AI accelerator on IBM Telum processor: industrial product". en. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York New York: ACM, June 2022, pp. 1012–1028. ISBN: 978-1-4503-8610-4. DOI: 10.1145/3470496.3533042. (Visited on 03/01/2023).

[Liu+21a] Xiao Fan Liu, Xin-Jian Jiang, Si-Hao Liu, and Chi Kong Tse. "Knowledge Discovery in Cryptocurrency Transactions: A Survey". In: *IEEE Access* 9 (2021), pp. 37229–37254. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3062652. URL: https://ieeexplore.ieee.org/document/9364978/ (visited on 04/04/2023).

[Liu+21b] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. "Pick and Choose: A GNN-Based Imbalanced Learning Approach for Fraud Detection". In: *Proceedings of the Web Conference 2021*. WWW '21. Ljubljana, Slovenia: Association for Computing Machinery, 2021, pp. 3168–3177. ISBN: 9781450383127. DOI: 10.1145/3442381.3449989. URL: https://doi.org/10.1145/3442381.3449989.

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. June 2014. URL: https://snap.stanford.edu/data (visited on 05/30/2022).

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. "Mining Social-Network Graphs". In: *Mining of Massive Datasets*. 2nd ed. Cambridge University Press, 2014, pp. 325–383. DOI: 10.1017/CBO9781139924801.011.

[LSL06] Guimei Liu, Kelvin Sim, and Jinyan Li. "Efficient Mining of Large Maximal Bicliques". In: *Data Warehousing and Knowledge Discovery*. Vol. 4081. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 437–448. ISBN: 978-3-540-37737-5. DOI: 10.1007/11823728_42. URL: http://link.springer.com/10.1007/11823728_42 (visited on 07/13/2021).

[LT82] G. Loizou and P. Thanisch. "Enumerating the cycles of a digraph: A new preprocessing strategy". en. In: *Information Sciences* 27.3 (Aug. 1982), pp. 163–182. ISSN: 00200255. DOI: 10.1016/0020-0255(82)90023-8.

[LW70] Don R. Lick and Arthur T. White. "k-Degenerate Graphs". en. In: *Can. j. math.* 22.5 (Oct. 1970), pp. 1082–1096. ISSN: 0008-414X, 1496-4279. DOI: 10.4153/CJM-1970-125-1. URL: https://www.cambridge.org/core/product/identifier/S0008414X00047854/type/journal_article (visited on 03/30/2023).

[LWN18] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. "Community Detection in Complex Networks via Clique Conductance". en. In: *Sci Rep* 8.1 (Dec. 2018), p. 5982. ISSN: 2045-2322. DOI: 10.1038/s41598-018-23932-z. URL: http://www.nature.com/articles/s41598-018-23932-z (visited on 08/21/2019).

[Mal+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: a system for large-scale graph processing”. en. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. Indianapolis Indiana USA: ACM, June 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184.

[Mat17] Nav Mathur. *Graph Technology for Financial Services*. Tech. rep. Accessed: 2022-05-30. Neo4J, 2017, pp. 1–14. URL: https://neo4j.com/use-cases/financial-services.

[MD76] Prabhaker Mateti and Narsingh Deo. “On Algorithms for Enumerating All Circuits of a Graph”. en. In: *SIAM J. Comput.* 5.1 (Mar. 1976), pp. 90–99. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0205007.

[MDR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming*. en. Elsevier, 2012. ISBN: 978-0-12-415993-8. DOI: 10.1016/C2011-0-04251-5. URL: https://linkinghub.elsevier.com/retrieve/pii/C20110042515 (visited on 03/25/2023).

[Meu+14] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. “Graph structure in the web — revisited: a trick of the heavy tail”. en. In: *Proceedings of the 23rd International Conference on World Wide Web - WWW ’14 Companion*. Seoul, Korea: ACM Press, 2014, pp. 427–432. ISBN: 978-1-4503-2745-9. DOI: 10.1145/2567948.2576928.

[MM65] J. W. Moon and L. Moser. “On cliques in graphs”. en. In: *Israel J. Math.* 3.1 (Mar. 1965), pp. 23–28. ISSN: 0021-2172, 1565-8511. DOI: 10.1007/BF02760024. URL: http://link.springer.com/10.1007/BF02760024 (visited on 03/26/2023).

[Mor+18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 9781931971478.

[MW19] Daniel Mawhirter and Bo Wu. “AutoMine: harmonizing high-level abstraction and high performance for graph mining”. en. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville Ontario Canada: ACM, Oct. 2019, pp. 509–523. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359633. URL: https://dl.acm.org/doi/10.1145/3341301.3359633 (visited on 08/30/2020).

[MWM15] Robert Ryan McCune, Tim Weninger, and Greg Madey. “Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing”. en. In: *ACM Comput. Surv.* 48.2 (Nov. 2015), pp. 1–39. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2818185.

[NKL21] Jack Nicholls, Aditya Kuppa, and Nhien-An Le-Khac. "Financial Cybercrime: A Comprehensive Survey of Deep Learning Approaches to Tackle the Evolving Financial Crime Landscape". In: *IEEE Access* 9 (2021), pp. 163965–163986. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3134076. URL: https://ieeexplore.ieee.org/document/9642993/ (visited on 04/05/2023).

[Noe+16] S. Noel, E. Harley, K.H. Tam, M. Limiero, and M. Share. "CyGraph: Graph-Based Analytics and Visualization for Cybersecurity". en. In: *Handbook of Statistics*. Vol. 35. Oxford, England: Elsevier, 2016, pp. 117–167. ISBN: 978-0-444-63744-4. DOI: 10.1016/bs.host.2016.07.001.

[Oli+18] Gabriele Oliva, Roberto Setola, Luigi Glielmo, and Christoforos N. Hadjicostis. "Distributed Cycle Detection and Removal". In: *IEEE Trans. Control Netw. Syst.* 5.1 (Mar. 2018), pp. 194–204. ISSN: 2325-5870. DOI: 10.1109/TCNS.2016.2593264. (Visited on 10/17/2020).

[Oli+21] Catarina Oliveira, João Torres, Maria Inês Silva, David Aparício, João Tiago Ascensão, and Pedro Bizarro. *GuiltyWalker: Distance to illicit nodes in the Bitcoin network*. arXiv:2102.05373 [cs]. July 2021. URL: http://arxiv.org/abs/2102.05373 (visited on 03/03/2023).

[Ora22] Oracle. *Java Documentation - Fork/Join*. Accessed: 2023-03-24. 2022. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[PA08] Girish Keshav Palshikar and Manoj M. Apte. "Collusion set detection using graph clustering". en. In: *Data Min Knowl Disc* 16.2 (Apr. 2008), pp. 135–164. ISSN: 1384-5810, 1573-756X. DOI: 10.1007/s10618-007-0076-8.

[Pag+98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Tech. rep. Stanford, CA, USA: Stanford Digital Library Technologies Project, Stanford University, Nov. 1998, p. 17. URL: https://web.archive.org/web/20110818093436/http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf (visited on 04/03/2023).

[PAS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: online learning of social representations". en. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York New York USA: ACM, Aug. 2014, pp. 701–710. ISBN: 978-1-4503-2956-9. DOI: 10.1145/2623330.2623732. URL: https://dl.acm.org/doi/10.1145/2623330.2623732 (visited on 03/03/2023).

[PBL17] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. "Motifs in Temporal Networks". en. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. Cambridge, United Kingdom: ACM, Feb. 2017, pp. 601–610. ISBN: 978-1-4503-4675-7. DOI: 10.1145/3018661.3018731.

[PD21] Sri Harsha Pothukuchi and Amit Dhuria. *Deterministic loop breaking in multi-mode multi-corner static timing analysis of integrated circuits*. Patent No. 11003821. May 2021.

[Pen+19] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. "Towards bridging theory and practice: hop-constrained s-t simple path enumeration". en. In: *Proc. VLDB Endow.* 13.4 (Dec. 2019), pp. 463–476. ISSN: 2150-8097. DOI: 10.14778/3372716.3372720.

[Pon66] J. Ponstein. "Self-Avoiding Paths and the Adjacency Matrix of a Graph". en. In: *SIAM J. Appl. Math.* 14.3 (Mar. 1966), pp. 600–609. ISSN: 0036-1399, 1095-712X. DOI: 10.1137/0114051.

[PR01] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo Hashing". In: *Algorithms — ESA 2001*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Friedhelm Meyer auf der Heide. Vol. 2161. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 121–133. ISBN: 978-3-540-44676-7. DOI: 10.1007/3-540-44676-1_10. URL: http://link.springer.com/10.1007/3-540-44676-1_10 (visited on 04/25/2020).

[Pri00] Erich Prisner. "Bicliques in Graphs I: Bounds on Their Number". In: *Combinatorica* 20.1 (Jan. 2000), pp. 109–117. ISSN: 0209-9683, 1439-6912. DOI: 10.1007/s004930070035. URL: http://link.springer.com/10.1007/s004930070035 (visited on 07/20/2021).

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. "Rethinking SIMD Vectorization for In-Memory Databases". en. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. Melbourne, Victoria, Australia: ACM Press, 2015, pp. 1493–1508. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2747645. URL: http://dl.acm.org/citation.cfm?doid=2723372.2747645 (visited on 02/23/2020).

[Qin+20] Zhu Qing, Long Yuan, Zi Chen, Jingjing Lin, and Guojie Ma. "Efficient Parallel Cycle Search in Large Graphs". en. In: *Database Systems for Advanced Applications*. Vol. 12113. Cham, Switzerland: Springer International, 2020, pp. 349–367. ISBN: 978-3-030-59416-9. DOI: 10.1007/978-3-030-59416-9{_}21.

[Qin+21] Xiao Qin, Nasrullah Sheikh, Berthold Reinwald, and Lingfei Wu. "Relation-aware Graph Attention Model with Adaptive Self-adversarial Training". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 11. May 2021, pp. 9368–9376. DOI: 10.1609/aaai.v35i11.17129. URL: https://ojs.aaai.org/index.php/AAAI/article/view/17129 (visited on 05/03/2023).

[Qiu+18] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. "Real-time constrained cycle detection in large dynamic graphs". en. In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 1876–1888. ISSN: 2150-8097. DOI: 10.14778/3229863.3229874.

[Qui04] Michael J. Quinn. *Parallel programming in C with MPI and openMP*. Dubuque, Iowa: McGraw-Hill, 2004. ISBN: 978-0-07-282256-4.

[RA12] Joseph Reddington and Kubilay Atasu. “Complexity of Computing Convex Subgraphs in Custom Instruction Synthesis”. In: *IEEE Trans. VLSI Syst.* 20.12 (Dec. 2012), pp. 2337–2341. ISSN: 1063-8210, 1557-9999. DOI: 10.1109/TVLSI.2011.2173221. URL: http://ieeexplore.ieee.org/document/6075311/ (visited on 07/11/2020).

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: http://networkrepository.com.

[Rao+21] Susie Xi Rao, Shuai Zhang, Zhichao Han, Zitao Zhang, Wei Min, Zhiyao Chen, Yinan Shan, Yang Zhao, and Ce Zhang. “xFraud: explainable fraud transaction detection”. en. In: *Proc. VLDB Endow.* 15.3 (Nov. 2021), pp. 427–436. ISSN: 2150-8097. DOI: 10.14778/3494124.3494128. (Visited on 01/09/2023).

[ref23a] C++ reference. *std::deque*. Accessed: 2023-02-21. 2023. URL: https://en.cppreference.com/w/cpp/container/deque.

[ref23b] C++ reference. *std::unordered_map*. Accessed: 2023-02-21. 2023. URL: https://en.cppreference.com/w/cpp/container/unordered_map.

[ROA13] Timothy G. Rogers, Mike O’Connor, and Tor M. Aamodt. “Divergence-Aware Warp Scheduling”. In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2013, pp. 99–110.

[Ron+21] Viktoria Ronge, Christoph Egger, Russell W. F. Lai, Dominique Schröder, and Hoover H. F. Yin. “Foundations of Ring Sampling”. en. In: *Proceedings on Privacy Enhancing Technologies* 2021.3 (July 2021), pp. 265–288. ISSN: 2299-0984. DOI: 10.2478/popets-2021-0047. URL: https://petsymposium.org/popets/2021/popets-2021-0047.php (visited on 04/04/2023).

[RT04] Peter Reuter and Edwin M. Truman. “Chasing Dirty Money: The Fight Against Money Laundering”. en. In: Washington, DC: Institute for International Economics, 2004. Chap. Money Laundering: Methods and Markets. ISBN: 978-0-88132-370-2. URL: https://www.piie.com/bookstore/chasing-dirty-money-fight-against-money-laundering (visited on 04/14/2023).

[RT15] Rodrigo Caetano Rocha and Bhalchandra D. Thatte. “Distributed cycle detection in large-scale sparse graphs”. en. In: *Simpósio Brasileiro de Pesquisa Operacional (SBPO)*. Porto de Galinhas, Pernambuco, Brasil: SOBRAPO, 2015, pp. 1–12. DOI: 10.13140/RG.2.1.1233.8640.

[RT75] R. C. Read and R. E. Tarjan. “Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees”. en. In: *Networks* 5.3 (July 1975), pp. 237–252. ISSN: 00283045. DOI: 10.1002/net.1975.5.3.237.

[Sak+16] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. en. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-47431-1. DOI: 10.1007/978-3-319-47431-1. URL: http://link.springer.com/10.1007/978-3-319-47431-1 (visited on 07/14/2020).

[SAS21] SAS. *SAS OPTGRAPH Procedure: Graph Algorithms and Network Analysis*. 2021. URL: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/procgralg/procgralg_optgraph_examples.htm (visited on 05/30/2022).

[Sch+09] Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park. "A scalable, parallel algorithm for maximal clique enumeration". en. In: *Journal of Parallel and Distributed Computing* 69.4 (Apr. 2009), pp. 417–428. ISSN: 07437315. DOI: 10.1016/j.jpdc.2009.01.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731509000082 (visited on 03/20/2019).

[Sch+21] Roman Schulte-Sasse, Stefan Budach, Denes Hnisz, and Annalisa Marsico. "Integration of multiomics data with graph convolutional networks to identify new cancer genes and their associated molecular mechanisms". In: *Nature Machine Intelligence* 3.6 (2021), pp. 513–526. DOI: 10.1038/s42256-021-00325-y. URL: https://doi.org/10.1038/s42256-021-00325-y.

[SEF16] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. "CoreScope: Graph Mining Using k-Core Analysis — Patterns, Anomalies and Algorithms". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. Barcelona, Spain: IEEE, Dec. 2016, pp. 469–478. ISBN: 978-1-5090-5473-2. DOI: 10.1109/ICDM.2016.0058. URL: http://ieeexplore.ieee.org/document/7837871/ (visited on 01/17/2020).

[Sew08] Julian Seward, ed. *Valgrind 3.3: advanced debugging and profiling for Gnu/Linux applications*. eng. 1. print. OCLC: 476326409. Bristol: Network Theory, 2008. ISBN: 978-0-9546120-5-4.

[SK21] Toyotaro Suzumura and Hiroki Kanezashi. *Anti-Money Laundering Datasets*. 2021. URL: https://github.com/IBM/AMLSim (visited on 05/30/2022).

[SL20] Shixuan Sun and Qiong Luo. "In-Memory Subgraph Matching: An In-depth Study". en. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, June 2020, pp. 1083–1098. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3380581. URL: https://dl.acm.org/doi/10.1145/3318464.3380581 (visited on 04/18/2021).

[SL76] J. Szwarcfiter and P. Lauer. "A search strategy for the elementary cycles of a directed graph". In: *BIT Numerical Mathematics* 16 (1976), pp. 192–204.

[SMT15] Michael Svendsen, Arko Provo Mukherjee, and Srikanta Tirthapura. "Mining maximal cliques from a large graph using MapReduce: Tackling highly uneven subproblem sizes". en. In: *Journal of Parallel and Distributed Computing* 79-80 (May 2015), pp. 104–114. ISSN: 07437315. DOI: 10.1016/j.jpdc.2014.08.011. URL:

https://linkinghub.elsevier.com/retrieve/pii/S0743731514001531 (visited on 07/12/2020).

[Sod15] Avinash Sodani. "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. Cupertino, CA, USA: IEEE, Aug. 2015, pp. 1–24. ISBN: 978-1-4673-8885-6. DOI: 10.1109/HOTCHIPS.2015.7477467. URL: http://ieeexplore.ieee.org/document/7477467/ (visited on 07/03/2019).

[Sta+21] Michele Starnini, Charalampos E. Tsourakakis, Maryam Zamanipour, André Panisson, Walter Allasia, et al. "Smurf-Based Anti-money Laundering in Time-Evolving Transaction Networks". en. In: *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track*. Vol. 12978. Cham: Springer International Publishing, 2021, pp. 171–186. ISBN: 978-3-030-86514-6. DOI: 10.1007/978-3-030-86514-6_11. (Visited on 01/10/2023).

[SWL11] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. "Fast Sorted-Set Intersection using SIMD Instructions". In: *ADMS@VLDB*. 2011.

[Tal+22] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, et al. "Mint: An Accelerator For Mining Temporal Motifs". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Chicago, IL, USA: IEEE, Oct. 2022, pp. 1270–1287. ISBN: 978-1-66546-272-3. DOI: 10.1109/MICRO56248.2022.00089. URL: https://ieeexplore.ieee.org/document/9923899/ (visited on 05/05/2023).

[Tar72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". en. In: *SIAM J. Comput.* 1.2 (June 1972), pp. 146–160. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0201010. URL: http://epubs.siam.org/doi/10.1137/0201010 (visited on 11/16/2020).

[Tar73] Robert Tarjan. "Enumeration of the Elementary Circuits of a Directed Graph". en. In: *SIAM J. Comput.* 2.3 (Sept. 1973), pp. 211–216. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0202017.

[Tie70] James C. Tiernan. "An efficient search algorithm to find the elementary circuits of a graph". en. In: *Commun. ACM* 13.12 (Dec. 1970), pp. 722–726. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362814.362819.

[TK12] Katharina Tschumitschew and Frank Klawonn. "Incremental Statistical Measures". en. In: *Learning in Non-Stationary Environments*. Ed. by Moamar Sayed-Mouchaweh and Edwin Lughofer. New York, NY: Springer New York, 2012, pp. 21–55. ISBN: 978-1-4419-8020-5. DOI: 10.1007/978-1-4419-8020-5_2. URL: http://link.springer.com/10.1007/978-1-4419-8020-5_2 (visited on 02/21/2023).

[TTT06] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. "The worst-case time complexity for generating all maximal cliques and computational experiments". en. In: *Theoretical Computer Science* 363.1 (Oct. 2006), pp. 28–42. ISSN: 03043975.

DOI: 10.1016/j.tcs.2006.06.015. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397506003586 (visited on 03/22/2019).

[Val90] Leslie G. Valiant. "A bridging model for parallel computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 00010782. DOI: 10.1145/79173.79181.

[VAR19] Michael Voss, Rafael Asenjo, and James Reinders. "Pro TBB: C++ Parallel Programming with Threading Building Blocks". en. In: Berkeley, CA: Apress, 2019. Chap. 16. ISBN: 978-1-4842-4398-5. DOI: 10.1007/978-1-4842-4398-5. URL: http://link.springer.com/10.1007/978-1-4842-4398-5 (visited on 06/16/2020).

[VBI10] Ajay K. Verma, Philip Brisk, and Paolo Ienne. "Fast, Nearly Optimal ISE Identification With I/O Serialization Through Maximal Clique Enumeration". In: *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 29.3 (Mar. 2010), pp. 341–354. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2010.2041849. URL: http://ieeexplore.ieee.org/document/5419242/ (visited on 07/11/2020).

[Vel+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. "Graph Attention Networks". In: *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=rJXMpikCZ.

[Wal21] Samourai Wallet. *Whirlpool Coinjoin*. 2021. URL: https://samouraiwallet.com/whirlpool (visited on 04/04/2023).

[Wan+20a] Fei Wang, Peng Cui, Jian Pei, Yangqiu Song, and Chengxi Zang. "Recent Advances on Graph Analytics and Its Applications in Healthcare". en. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Virtual Event CA USA: ACM, 2020, pp. 3545–3546. ISBN: 978-1-4503-7998-4. DOI: 10.1145/3394486.3406469.

[Wan+20b] Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. "Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 173–186. ISBN: 978-1-72814-661-4. DOI: 10.1109/ISCA45697.2020.00025. URL: https://ieeexplore.ieee.org/document/9138949/ (visited on 10/02/2020).

[Wan+21] Jianian Wang, Sheng Zhang, Yanghua Xiao, and Rui Song. "A Review on Graph Neural Network Methods in Financial Applications". In: *CoRR* abs/2111.15367 (2021). arXiv: 2111.15367. URL: https://arxiv.org/abs/2111.15367.

[Web+18] Mark Weber, Jie Chen, Toyotaro Suzumura, Aldo Pareja, Tengfei Ma, Hiroki Kanezashi, Tim Kaler, Charles E. Leiserson, and Tao B. Schardl. "Scalable Graph Learning for Anti-Money Laundering: A First Look". In: *arXiv:1812.00076 [cs]* (Nov. 2018). arXiv: 1812.00076. URL: http://arxiv.org/abs/1812.00076 (visited on 07/11/2021).

[Web+19] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I Weidele, Claudio Bellei, Tom Robinson, and Charles E Leiserson. "Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics". In: *arXiv preprint arXiv:1908.02591* (2019).

[Web21] Jim Webber. *Powering Real-Time Recommendations with Graph Database Technology*. Tech. rep. Accessed: 2022-05-30. Neo4J, 2021, pp. 1–7. URL: https://neo4j.com/use-cases/real-time-recommendation-engine.

[Wei72] Herbert Weinblatt. "A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph". en. In: *J. ACM* 19.1 (Jan. 1972), pp. 43–56. ISSN: 0004-5411, 1557-735X.

[Wel65] J. T. Welch. "Numerical applications: Cycle algorithms for undirected linear graphs and some immediate applications". en. In: *Proceedings of the 1965 20th National Conference*. Cleveland, Ohio, United States: ACM Press, 1965, pp. 296–301. DOI: 10.1145/800197.806053.

[Woe99] Philipp Woelfel. "Efficient Strongly Universal and Optimally Universal Hashing". In: *Mathematical Foundations of Computer Science 1999*. Vol. 1672. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 262–272. ISBN: 978-3-540-48340-3. DOI: 10.1007/3-540-48340-3_24. URL: http://link.springer.com/10.1007/3-540-48340-3_24 (visited on 06/24/2020).

[Wu+21] Jiajing Wu, Jieli Liu, Weili Chen, Huawei Huang, Zibin Zheng, and Yan Zhang. "Detecting Mixing Services via Mining Bitcoin Transaction Network With Hybrid Motifs". In: *IEEE Trans. Syst. Man Cybern, Syst.* (2021), pp. 1–13. ISSN: 2168-2216, 2168-2232. DOI: 10.1109/TSMC.2021.3049278. URL: https://ieeexplore.ieee.org/document/9332279/ (visited on 11/10/2021).

[Xu+14] Yanyan Xu, James Cheng, Ada Wai-Chee Fu, and Yingyi Bu. "Distributed Maximal Clique Computation". In: *2014 IEEE International Congress on Big Data*. Anchorage, AK, USA: IEEE, June 2014, pp. 160–167. DOI: 10.1109/BigData.Congress.2014.31. URL: http://ieeexplore.ieee.org/document/6906774/ (visited on 01/16/2020).

[Xu+18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018).

[Yao+20] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. "A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Athens, Greece: IEEE, Oct. 2020, pp. 895–907. ISBN: 978-1-72817-383-2. DOI: 10.1109/MICRO50266.2020.00077. URL: https://ieeexplore.ieee.org/document/9251861/ (visited on 12/15/2020).

[Yu+06] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. "Predicting interactions in protein networks by completing defective cliques". en. In: *Bioinformatics* 22.7 (Apr. 2006), pp. 823–829. ISSN: 1460-2059, 1367-4803. DOI: 10.1093/bioinformatics/btl014. URL: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btl014 (visited on 03/05/2020).

[YZT14] Lei Yang, Xudong Zhao, and Xianglong Tang. "Predicting Disease-Related Proteins Based on Clique Backbone in Protein-Protein Interaction Network". en. In: *Int. J. Biol. Sci.* 10.7 (2014), pp. 677–688. ISSN: 1449-2288. DOI: 10.7150/ijbs.8430. URL: http://www.ijbs.com/v10p0677.htm (visited on 03/05/2020).

[Zha+21] Zaixi Zhang, Qi Liu, Hao Wang, Chengqiang Lu, and Cheekong Lee. "Motif-based Graph Self-Supervised Learning for Molecular Property Prediction". In: *CoRR* abs/2110.00987 (2021). arXiv: 2110.00987. URL: https://arxiv.org/abs/2110.00987.

[Zho+18] Xiaoping Zhou, Xun Liang, Jichao Zhao, and Shusen Zhang. "Cycle Based Network Centrality". en. In: *Sci Rep* 8.1 (Dec. 2018), p. 11749. ISSN: 2045-2322. DOI: 10.1038/s41598-018-30249-4.

[Zhu+20a] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. "LiveGraph: a transactional graph storage system with purely sequential adjacency list scans". en. In: *Proc. VLDB Endow.* 13.7 (Mar. 2020), pp. 1020–1034. ISSN: 2150-8097. DOI: 10.14778/3384345.3384351. URL: https://dl.acm.org/doi/10.14778/3384345.3384351 (visited on 02/28/2023).

[Zhu+20b] Yongchun Zhu, Dongbo Xi, Bowen Song, Fuzhen Zhuang, Shuai Chen, Xi Gu, and Qing He. "Modeling Users' Behavior Sequences with Hierarchical Explainable Network for Cross-domain Fraud Detection". en. In: *Proceedings of The Web Conference 2020*. Taipei Taiwan: ACM, Apr. 2020, pp. 928–938. ISBN: 978-1-4503-7023-3. DOI: 10.1145/3366423.3380172. (Visited on 01/10/2023).

JOVAN BLANUŠA

www.linkedin.com/in/jovan-blanusa

EDUCATION

Lausanne, Switzerland **École Polytechnique Fédérale de Lausanne (EPFL)** **October 2018 – August 2023**

- *PhD in Computer and Communication Sciences.*
- Advisors: Prof. Paolo Ienne and Dr. Kubilay Atasu.
- Thesis: Acceleration of graph pattern mining and applications to financial crime.

Lausanne, Switzerland **École Polytechnique Fédérale de Lausanne (EPFL)** **September 2016 – July 2018**

- *Master of Science in Electrical and Electronics Engineering.*
- Orientation: Electronics and microelectronics; GPA: 5.64/6
- Holder of the Excellence Fellowship awarded for outstanding academic records.

Belgrade, Serbia **University of Belgrade** **October 2012 – September 2016**

- *Bachelor of Science in Electrical Engineering and Computing.*
- Orientation: Electronics; GPA: 10/10
- Honor for the best student of the generation that graduated in 2016.

Novi Sad, Serbia **Gymnasium Jovan Jovanović Zmaj** **September 2008 – May 2012**

- *The class for students gifted in mathematics, physics, and computer science.*
- Honor for the best student of the graduating class.

WORKING EXPERIENCE

IBM Research Europe **Zurich, Switzerland**

- *Postdoctoral researcher* *July 2023 – present*
  - Part of the *Infrastructure AIOPS team* in the *Hybrid Cloud department*.
  - Focus on developing real-time graph-based solutions for the detection of suspicious financial transactions.
- *Predoctoral researcher* *April 2019 – June 2023*
  - Developed state-of-the-art parallel algorithms for clique (VLDB'20) and cycle enumeration (TOPC'23).
  - Developed a real-time graph-based feature extraction library that can significantly increase the accuracy of machine learning models for detecting financial crime.
  - The library has been integrated into IBM's Snap ML library, it is available in PyPI, and it is offered as part of IBM Cloud Pak for Data (CP4D 4.6.3).

Microelectronic Systems Laboratory, EPFL **Lausanne, Switzerland**

- *Research assistant* *October 2018 – March 2019*
  Researcher in the lab working on development and implementation of high-speed integrated circuits for camera systems.

NVIDIA **Santa Clara, California, USA**

- *Hardware Verification Intern* *July 2018 – September 2018*
  Part of the HW ASIC GPU team, working on post silicon performance tests for NVIDIA's proprietary GPU interconnect NVLINK.

NVIDIA **Santa Clara, California, USA**

- *System Software Engineer Intern* *September 2017 – December 2017*
  Part of the camera software team for Shield TV, working on HAL-based USB camera test cases for Android O.

Microsoft **Belgrade, Serbia**

- *Intern Software Design Engineer* *February 2015 – June 2015*
  Part of SQL Team, working on service integration prototype for Microsoft Azure web services.

PUBLICATIONS AND PATENTS

- J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "**Parallelizing Maximal Clique Enumeration on Modern Manycore Processors**", *IEEE IPDPSW 2020*, New Orleans, LA, USA, May 2020, pp. 211–214.
- J. Blanuša, R. Stoica, P. Ienne, and K. Atasu. "**Manycore Clique Enumeration with Fast Set Intersections**", *PVLDB, 13(12)*, Tokyo, Japan, August 2020, pp. 2676-2690.
- J. Blanuša, P. Ienne, and K. Atasu. "**Scalable Fine-Grained Parallel Cycle Enumeration Algorithms**", *ACM SPAA 2022,* Philadelphia, PA, USA, July 2022, pp. 247–258.
- J. Blanuša, K. Atasu, and P. Ienne. "**Fast Parallel Algorithms for Enumeration of Simple, Temporal, and Hop-Constrained Cycles**", To appear in *ACM Transactions on Parallel Computing*, 2023.
- J. Blanuša, M. Cravero, K. Atasu, and H. Pozidis. "**Processing Graphs using Graph Patterns**", IBM Patent application filed with the US patent office under application number 18/194701.
- B. Egressy, L. von Niederhäusern, J. Blanuša, E. Altman, R, Wattenhofer, and K. Atasu. "**Provably Powerful Graph Neural Networks for Directed Multigraphs**", preprint, *arXiv:2306.11586*, June 2023.
- E. Altman, B. Egressy, J. Blanuša, and K. Atasu. 2023. "**Realistic Synthetic Financial Transactions for Anti-Money Laundering Models**", preprint, *arXiv:2306.16424*, June 2023.

OPEN-SOURCE AND PUBLICLY AVAILABLE LIBRARIES

- **Parallel Clique Enumeration** – open-source project
  - Source code: https://github.com/IBM/parallel-clique-enumeration
- **Parallel Cycle Enumeration** – open-source project
  - Source code: https://github.com/IBM/parallel-cycle-enumeration
- **Graph Feature Preprocessor** – publicly available software library
  - Library: https://pypi.org/project/snapml/
  - Documentation: https://snapml.readthedocs.io/en/latest/graph_preprocessor.html

HONORS AND AWARDS

EPFLinnovators PhD Fellowship — **September 2018**
- Awarded to excellent PhD candidates interested in entrepreneurship — *Lausanne, Switzerland*

The student of the generation 2016 — **September 2017**
- Awarded to the best students of each school at the University of Belgrade — *Belgrade, Serbia*

EPFL Excellence Fellowship — **March 2016**
- Awarded for the outstanding academic record prior to admission — *Lausanne, Switzerland*

43rd International Physics Olympiad — **July 2012**
- Bronze medal — *Tallinn and Tartu, Estonia*

6th Serbian Physics Olympiad of high school students — **May 2012**
- Second award — *Belgrade, Serbia*

6th Serbian Mathematics Olympiad of high school students — **April 2012**
- Third award — *Belgrade, Serbia*

SKILLS

- **Programming languages and HDL:** C, C++, Python, MATLAB, Bash scripting, VHDL, Verilog
- **Tools and technologies**: Intel TBB, OpenMP, MPI, Vector instructions (AVX), Intel VTune profiler, FPGA development, Altera Quartus, Xilinx Vivado, Xilinx Vitis, Cadence Virtuoso
- **Other**: Git, LaTeX, Linux, MS Excel
- **Language proficiency:** English (fluent), Serbian (native), German (basic)